Learning Scientific Programming with Python

Learn to master basic programming tasks from scratch with real-life, scientifically relevant examples and solutions drawn from both science and engineering. Students and researchers at all levels are increasingly turning to the powerful Python programming language as an alternative to commercial packages and this fast-paced introduction moves from the basics to advanced concepts in one complete volume, enabling readers to quickly gain proficiency.

Beginning with general programming concepts such as loops and functions within the core Python 3 language, and moving on to the NumPy, SciPy and Matplotlib libraries for numerical programming and data visualization, this textbook also discusses the use of IPython Notebooks to build rich-media, shareable documents for scientific analysis. Including a final chapter introducing challenging topics such as floating-point precision and algorithm stability, and with extensive online resources to support advanced study, this textbook represents a targeted package for students requiring a solid foundation in Python programming.

Christian Hill is a physicist and physical chemist at University College London and Oxford University. He has over twenty years' experience of programming in the physical sciences and has been programming in Python for ten years. His research uses Python to produce, analyse, process, curate and visualize large data sets for the prediction of the properties of planetary atmospheres.

Learning Scientific Programming with Python

Christian Hill

University College London and
Somerville College, University of Oxford

CAMBRIDGE
UNIVERSITY PRESS

CAMBRIDGE
UNIVERSITY PRESS

University Printing House, Cambridge CB2 8BS, United Kingdom

Cambridge University Press is part of the University of Cambridge.

It furthers the University's mission by disseminating knowledge in the pursuit of education, learning and research at the highest international levels of excellence.

www.cambridge.org
Information on this title: www.cambridge.org/9781107075412

First published 2015
3rd printing 2018

Printed in the United Kingdom by TJ International, Padstow, Cornwall

A catalogue record for this publication is available from the British Library

Library of Congress Cataloguing in Publication data
Hill, Christian, 1974–
Learning scientific programming with Python / Christian Hill,
University College London and Somerville College, University of Oxford.
pages cm
ISBN 978-1-107-07541-2 (Hardback) – ISBN 978-1-107-42822-5 (Paperback)
1. Science–Data processing. 2. Science–Mathematics.
3. Python (Computer program language) I. Title.
Q183.9.H58 2015
005.13′3–dc23 2015017085

ISBN 978-1-107-07541-2 Hardback
ISBN 978-1-107-42822-5 Paperback

Additional resources for this publication at www.cambridge.org/9781107075412

Contents

Acknowledgments *page* vii

1 Introduction 1

1.1 About this book 1
1.2 About Python 2
1.3 Installing Python 5
1.4 The command line 6

2 The core Python language I 8

2.1 The Python shell 8
2.2 Numbers, variables, comparisons and logic 9
2.3 Python objects I: strings 27
2.4 Python objects II: lists, tuples and loops 41
2.5 Control flow 56
2.6 File input/output 66
2.7 Functions 70

3 Interlude: simple plotting with `pylab` 84

3.1 Basic plotting 84
3.2 Labels, legends and customization 89
3.3 More advanced plotting 97

4 The core Python language II 102

4.1 Errors and exceptions 102
4.2 Python objects III: dictionaries and sets 110
4.3 Pythonic idioms: “syntactic sugar” 121
4.4 Operating system services 131
4.5 Modules and packages 137
4.6 An introduction to object-oriented programming 147

5 IPython and IPython Notebook 160

5.1 IPython 160
5.2 IPython Notebook 174

6 NumPy 184

6.1 Basic array methods 184
6.2 Reading and writing an array to a file 216
6.3 Statistical methods 225
6.4 Polynomials 232
6.5 Linear algebra 247
6.6 Matrices 256
6.7 Random sampling 262
6.8 Discrete Fourier transforms 272

7 Matplotlib 280

7.1 Matplotlib basics 280
7.2 Contour plots, heatmaps and 3D plots 317

8 SciPy 333

8.1 Physical constants and special functions 333
8.2 Integration and ordinary differential equations 355
8.3 Interpolation 374
8.4 Optimization, data-fitting and root-finding 380

9 General scientific programming 402

9.1 Floating point arithmetic 402
9.2 Stability and conditioning 410
9.3 Programming techniques and software development 415

Appendix A Solutions 424

Index 445

Acknowledgments

For Emma, Charlotte and Laurence

Many people have helped directly or indirectly in the preparation of this book. Thanks are due especially to Jonathan Tennyson at UCL, and Laurence Rothman and Iouli Gordon for hosting my sabbatical year at the Harvard-Smithsonian Center for Astrophysics. As ever, I owe much to Natalie Haynes for her constant support, encouragement and friendship.

1 Introduction

1.1 About this book

This book is intended to help scientists and engineers learn version 3 of the Python programming language and its associated NumPy, SciPy and Matplotlib libraries. No prior programming experience or scientific knowledge in any particular field is assumed. However, familiarity with some mathematical concepts such as trigonometry, complex numbers and basic calculus is helpful to follow the examples and exercises.

Python is a powerful language with many advanced features and libraries; while the basic syntax of the language is straightforward to learn, it would be impossible to teach it in depth in a book of this size. Therefore, we aim for a balanced, broad introduction to the central features of the language and its important libraries. The text is interspersed with examples relevant to scientific research, and at the end of most sections there are questions (short problems designed to test knowledge) and exercises (longer problems that usually require a short computer program to solve). Although it is not necessary to complete all of the exercises, readers will find it useful to attempt at least some of them. Where a section, example or exercise contains more advanced material that may be skipped on first reading, this is indicated with the symbol ♢.

In Chapter 2 of this book, the basic syntax, data structures and flow control of a Python program are introduced. Chapter 3 is a short interlude on the use of the Pylab library for making graphical plots of data: this is useful to visualize the output of programs in subsequent chapters. Chapter 4 provides more advanced coverage of the core Python language and a brief introduction to object-oriented programming. There follows another short chapter introducing the popular IPython and IPython Notebook environments, before chapters on scientific programming with NumPy, Matplotlib and SciPy. The final chapter covers more general topics in scientific programming, including floating point arithmetic, algorithm stability and programming style.

Readers who are already familiar with the Python programming language may wish to skim Chapters 2 and 4.

Code examples and exercise solutions may be downloaded from the book's website at scipython.com. Note that while comments have been included in these downloadable programs, they are not so extensive in the printed version of this book: instead, the code is explained in the text itself through numbered annotations (such as ❶). Readers typing in these programs for themselves may wish to add their own explanatory comments to the code.

1.2 About Python

Python is a powerful, general-purpose programming language devised by Guido van Rossum in 1989.[1] It is classified as a high-level programming language in that it automatically handles the most fundamental operations (such as memory management) carried out at the processor level ("machine code"). It is considered a higher-level language than, for example, C, because of its expressive syntax (which is close to natural language in some cases) and rich variety of native data structures such as lists, tuples, sets and dictionaries. For example, consider the following Python program which outputs a list of names on separate lines.

Listing 1.1 Outputing a list of names using a program written in Python

```
# eg1-names.py: output three names to the console.

names = ['Isaac Newton', 'Marie Curie', 'Albert Einstein']
for name in names:
    print(name)
```

Output:

```
Isaac Newton
Marie Curie
Albert Einstein
```

Now compare this with the equivalent program in C.

Listing 1.2 Outputing a list of names using a program written in C

```
/* eg1-names.c: output three names to the console. */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define MAX_STRING_LENGTH 20
#define NUMBER_OF_STRINGS 3

int main()
{
int i;
char names[NUMBER_OF_STRINGS][MAX_STRING_LENGTH+1];
strcpy(names[0], "Isaac Newton");
strcpy(names[1], "Marie Curie");
strcpy(names[2], "Albert Einstein");

for (i=0;i<NUMBER_OF_STRINGS;i++) {
    fprintf(stdout, "%s\n", names[i]);
}

return EXIT_SUCCESS;
}
```

[1] Python's "benevolent dictator for life."

Even if you are not familiar with the C language, you can see there is quite a lot of overhead involved in coding even this simple task in C: three `includes` of libraries not loaded by default, explicit declarations of variables to hold the list ("array", in C) of names, `names`, and a counter, `i`, and explicit indexing of this array in a `for` loop; you even need to add the line endings ('`\n`' is the "new line" character). This source code then has to be *compiled* – converted into the machine code that the computer processor understands – before it can be run (*executed*). Furthermore, there is plenty of scope for errors (bugs): trying to print the name stored in `name[10]` will likely cause junk to be output: the C compiler won't stop you from accessing this nonexistent name.

The same program written in three lines of Python is clean and expressive: we do not have to explicitly declare that `names` is a list of strings, there is no need for a loop counter like `i` and there are no separate libraries to include (`import` in Python). To run the Python program, one simply needs to type `python eg1-names.py` which will automatically invoke the Python "interpreter" to compile and then run the resulting "byte-code" (a kind of intermediate representation of the program between its source and the ultimate machine code that Python dispatches to the processor).

Python's syntax aims to ensure that "There should be one – and preferably only one – obvious way to do it." This differs from some other popular high-level languages such as Ruby and Perl, which take the opposite approach, encapsulated by the mantra "there's more than one way to do it." For example, there are (at least) four obvious ways to output the same list in Perl:[2]

Listing 1.3 Different ways to output a list of names using a program written in Perl

```
@names = ("Isaac Newton", "Marie Curie", "Albert Einstein");
# Method 1
print "$_\n" for @names;

# Method 2
print join "\n", @names;
print "\n";

# Method 3
print map { "$_\n" } @names;

# Method 4
$" = "\n";
print "@names\n";
```

(Note also Perl's famously concise but somewhat opaque syntax.)

1.2.1 Advantages and disadvantages of Python

Here are some of the main advantages of the Python programming language and why you might want to use it:

[2] Well, obvious to Perl programmers.

- Its clean and simple syntax makes writing Python programs fast and generally minimizes opportunities for bugs to creep in. When done right, the result is high-quality software that is easy to maintain and extend.
- It's free – Python and its associated libraries are free of cost and open source, unlike commercial offerings such as Mathematica.
- Cross-platform support: Python is available for every commonly available computer system, including Windows, Unix, Linux and Mac OS X. Although platform-specific extensions exist, it is possible to write code that will run on any platform without modification.
- Python has a large library of modules and packages that extend its functionality. Many of these are available as part of the "standard library" provided with the Python interpreter itself. Others, including the NumPy, SciPy and Matplotlib libraries used in scientific computing, can be downloaded separately for no cost.
- Python is relatively easy to learn. The syntax and idioms used for basic operations are applied consistently in more advanced usage of the language. Error messages are generally meaningful assessments of what went wrong rather than the generic "crashes" that can occur in compiled lower-level languages such as C.
- Python is flexible: it is often described as a "multi-paradigm" language that contains the best features from the procedural, object-oriented and functional programming paradigms. There is little need for the work-arounds required in some languages when a problem can only be solved cleanly with one of these approaches.

So where's the catch? Well, Python does have some disadvantages and isn't suitable for every application.

- The speed of execution of a Python program is not as fast as some other, fully compiled languages such as C and Fortran. For heavily numerical work, the NumPy and SciPy libraries alleviate this to some extent by using compiled-C code "under the hood," but at the expense of some reduced flexibility. For many, many applications, however, the speed difference is not noticeable and the reduced speed of execution more than offset by a much faster speed of *development*. That is, it takes much less time to write and debug a Python program than to do the same in C, C++ or Java.
- It is hard to hide or obfuscate the source code of a Python program to prevent others from copying or modifying it. However, this doesn't mean that successful commercial Python programs don't exist.
- A common complaint about Python has historically been that its rapid development has led to compatibility issues between versions. Certainly there are important differences between Python 2 and Python 3 (described in the next section), but the complaint stems from the fact that within the Python 2 series, there were major improvements and additions to the language that meant that code written in a later version (say, 2.7) would not run on an earlier version of Python (e.g., 2.6), although code written for an earlier version of Python will always run on a later version (within the same branch, 2 or 3). If you use the

latest version of Python (see Section 1.3) you probably won't run into a problem, but some operating systems that come with Python are rather conservative and install by default only an older version.

1.2.2 Python 2 or Python 3?

At the time of writing, Python users have a choice to make: whether to use the older, more established Python 2 version of the language or the newer Python 3. Although the differences between the two versions may seem minor, code written in Python 3 will not run under Python 2 and vice versa: Python 3 is not *backward-compatible* with its predecessor. **This book teaches Python 3**.

The latest major version of Python 2, Python 2.7, will be the last of that branch. Since its release in 2009, the number of users and extent of library support for Python 3 has grown to the point that new users would find little benefit in learning Python 2 except to maintain legacy code.

There are several reasons for major change between versions (breaking your users' existing code is not something to be undertaken lightly): Python 3 fixes some ugly quirks and inconsistencies in the language and provides *Unicode support* for all strings (eliminating a lot of the confusion that is created in dealing with Unicode and non-Unicode strings in Python 2). Unicode is an international standard for the representation of text in most of the writing systems in the world.

It is anticipated that most users of this book will not have trouble converting their own code between the two versions of Python if necessary. Where important, the differences are pointed out in the text.

1.3 Installing Python

The official website of Python is www.python.org/, and contains full and easy-to-follow instructions for downloading Python. However, there are several full distributions which include the NumPy, SciPy and Matplotlib libraries (the "SciPy stack") to save you from having to download and install these yourself:

- *Anaconda* is available for free (including for commerical use) from http://continuum.io/downloads. It installs both Python 2 and Python 3, but the default version can be selected either before downloading as indicated on this web page, or subsequently using the '`conda`' command.
- *Enthought Canopy* is a similar distribution with a free version and various tiers of paid-for versions including technical support and development software.

In most cases, one of these distributions should be all you need. We provide some platform-specific notes below.

The source code (and binaries for some platforms) for the NumPy, SciPy, Matplotlib and IPython packages are available separately at:

- NumPy: http://sourceforge.net/projects/numpy/
- SciPy: http://sourceforge.net/projects/scipy/
- Matplotlib: http://matplotlib.org/downloads.html
- IPython: https://github.com/ipython/ipython/releases

Windows

Windows users have a couple of further options for installing the full SciPy stack: *Python(x,y)* (https://code.google.com/p/pythonxy/) and *WinPython* (http://winpython.sourceforge.net/). Both are free.

Mac OS X

Mac OS X, being based on Unix, comes with Python, but it is usually an older version of Python 2. You must not delete or modify this installation (it's needed by the operating system), but you can follow the instructions above for obtaining Python 3 and the SciPy stack. OS X does not have a native *package manager* (an application for managing and installing software), but the two popular third-party package managers, Homebrew (http://brew.sh/) and MacPorts (www.macports.org/), can both supply Python 3 and its packages if you prefer this option.

Linux

Almost all Linux distributions come with Python 2, but usually not Python 3, so you will need to install it from the links above: the Anaconda and Canopy distributions both have versions for Linux. Most Linux distributions come with their own software package managers (e.g., `apt` in Debian and `rpm` for RedHat). These can be used to install Python 3 and its libraries, though finding the necessary package repositories may take some research on the Internet. Be careful not to replace or modify your system installation as other applications may depend on it.

1.4 The command line

Most of the code examples in this book are written as standalone programs which can be run from the *command line* (or from within an *integrated development environment* (IDE) if you use one: see Section 9.3.2). To access the command line interface (also known as a console or terminal) on different platforms, follow the instructions below.

- Windows 7 and earlier: *Start* > *All Programs* > *Command Prompt*; alternatively, type `cmd` in the *Start* > *Run* input box.
- Windows 8: *Preview* (lower left of screen) > *Windows System: All apps*; alternatively type '`cmd`' in the search box pulled down the top-right corner of the screen.
- Mac OS X: *Finder* > *Applications* > *Utilities* > *Terminal*
- Linux: if you are not using a graphical interface you are already at the command line; if you are, then locate the Terminal application (distributions vary, but it is usually found within a *System Utilities* or *System Tools* subfolder).

Commands typed at the command line are interpreted by an application called a *shell*, which allows the user to navigate the file system and is able to start other applications. For example, the command

```
python myprog.py
```

instructs the shell to invoke the Python interpreter, sending it the file `myprog.py` as the script to execute. Output from the program is then returned to the shell and displayed in your console.

2 The core Python language I

2.1 The Python shell

This chapter introduces the syntax, structure and data types of the Python programming language. The first few sections do not involve writing much beyond a few statements of Python code and so can be followed using the Python *shell*. This is an interactive environment: the user enters Python statements that are executed immediately after the *Enter* key is pressed.

The steps for accessing the "native" Python shell differ by operating system. To start it from the command line, first open a terminal using the instructions from Section 1.4 and type `python`.

To exit the Python shell, type `exit()`.

When you start the Python shell, you will be greeted by a message (which will vary depending on your operating system and precise Python version). On my system, the message reads:

```
Python 3.3.5 |Anaconda 2.0.1 (x86\_64)| (default, Mar 10 2014, 11:22:25)
[GCC 4.0.1 (Apple Inc. build 5493)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

The three chevrons (`>>>`) are the *prompt*, which is where you will enter your Python commands. Note that this book is concerned with Python 3, so you should check that the Python version number reported on the first line is `Python 3.X.Y` where the precise values of the minor version numbers `X` and `Y` should not be important.

Many Python distributions come with a slightly more advanced shell called *IDLE*, which features tab-completion, and *syntax highlighting* (Python keywords are colored specially when you type them). We will pass over the use of this application in favor of the newer and more advanced *IPython* environment, discussed in Chapter 5.

It is also possible for many installations (especially on Windows) to start a Python shell directly from an application installed when you install the Python interpreter itself. Some installations even add a shortcut icon to your Desktop which will open a Python shell when you click on it.

2.2 Numbers, variables, comparisons and logic

2.2.1 Types of numbers

Among the most basic Python objects are the numbers, which come in three *types*: integers (type: `int`), floating point numbers (type: `float`) and complex numbers (type: `complex`).

Integers

Integers are whole numbers such as 1, 8, −72 and 3847298893721407. In Python 3, there is no limit to their magnitude (apart from the availability of your computer's memory).[1] Integer arithmetic is exact.

Floating point numbers

Floating point numbers are the representation of real numbers such as 1.2, −0.36 and 1.67263×10^{-7}. They do not, in general, have the exact value of the real number they represent, but are stored in binary to a certain precision (on most systems, to the equivalent of 15–16 decimal places),[2] as explained in Section 9.1. For example, the number $\frac{4}{3}$ is stored as the binary equivalent of $1.3333333333333332593184650 2\cdots$, which is nearly (but not quite) the same as the infinitely repeating decimal representation of $\frac{4}{3} = 1.3333\cdots$. Moreover, even numbers that do have an exact *decimal* representation may not have an exact *binary* representation: for example $1/10$ is represented by the binary number equivalent to $0.1000000000000000055511151 2\cdots$. Because of this finite precision, floating point arithmetic is not exact but, with care, it is "good enough" for most scientific applications.

Any single number containing a period ('`.`') is considered by Python to specify a floating point number. Scientific notation is supported using '`e`' or '`E`' to separate the significand (mantissa) from the exponent: for example, `1.67263e-7` represents the number 1.67263×10^{-7}.

Complex numbers

Complex numbers such as $4+3j$ consist of a real and an imaginary part (denoted by `j` in Python), each of which is itself represented as a floating point number (even if specified without a period). Complex number arithmetic is therefore not exact but subject to the same finite precision considerations as `float`s.

A complex number may be specified either by "adding" a real number to an imaginary one (denoted by the `j` suffix), as in `2.3 + 1.2j` or by separating the real and imaginary parts in a call to `complex`, as in `complex(2.3, 1.2)`.

[1] In Python 2, there were two kinds of integer: "simple" integers (system-dependent, but usually stored in either 32 or 64 bits and "long" integers (of any size), indicated with the suffix `L`.

[2] This corresponds to the implementation of the IEEE 754 double-precision standard.

Example E2.1 Typing a number at the Python shell prompt simply echoes the number back to you:

```
>>> 5
5
>>> 5.
5.0
>>> 0.10
❶ 0.1
>>> 0.0001
0.0001
>>> 0.0000999
❷ 9.99e-05
```

Note that the Python interpreter displays numbers in a standard way. For example:

❶ The internal representation of 0.1 discussed earlier is rounded to '`0.1`', which is the shortest number with this representation.

❷ Numbers smaller in magnitude than 0.0001 are displayed in scientific notation.

A number of one type can be created from a number of another type with the relevant *constructor*:

```
>>> float(5)
5.0
>>> int(5.2)
5
>>> int(5.9)
❶ 5
>>> complex(3.)
❷ (3+0j)
❸ >>> complex(0., 3.)
3j
```

❶ Note that a floating point number is *rounded down* in casting it into an integer.

❷ Constructing a `complex` object from a `float` generates a complex number with the imaginary part equal to zero.

❸ To generate a pure imaginary number, you have to explicitly pass two numbers to `complex` with the first, real part, equal to zero.

2.2.2 Using the Python shell as a calculator

Basic arithmetic

With the three basic number types described earlier, it is possible to use the Python shell as a simple calculator using the operators given in Table 2.1. These are *binary* operators in that they act on two numbers (the *operands*) to produce a third (e.g., `2**3` evaluates to `8`).

Python 3 has two types of division: floating point division (`/`) always returns a floating point (or complex) number result, even if it acts on integers. Integer division (`//`) *always rounds down* the result to the nearest integer; the type of the resulting number is an `int`

Table 2.1 Basic Python arithmetic operators

`+`	addition
`-`	subtraction
`*`	multiplication
`/`	floating point division
`//`	integer division
`%`	modulus (remainder)
`**`	exponentiation

only if both of its operands are `int`s; otherwise it returns a `float`. Some examples should make this clearer:

Regular floating point division with (`/`):

```
>>> 2.7 / 2
1.35
>>> 9 / 2
4.5
>>> 8 / 4
2.0
```

The last operation returns a `float` even though both operands are `int`s.[3]

Integer division with (`//`):

```
>>> 8 // 4
2
>>> 9 // 2
4
>>> 2.7 // 2
1.0
```

Note that `//` can perform integer arithmetic (rounding down) on floating point numbers.

The modulus operator gives the remainder of an integer division:

```
>>> 9 % 2
1
>>> 4.5 % 3
1.5
```

Again, the number returned is an `int` only if both of the operands are `int`s.

Operator precedence

Arithmetic operations can be strung together in a sequence, which naturally raises the question of *precedence*: for example, does `2 + 4 * 3` evaluate to `14` (as `2 + 12`) or `18` (as `6 * 3`)? Table 2.2 shows that the answer is `14`: multiplication has a higher precedence than addition and is evaluated first. These precedence rules are overridden by the use of *parentheses*: for example, `(2 + 4) * 3 = 18`.

[3] This is a major difference from Python 2, in which the `/` operator performed integer division on two integers.

Table 2.2 Python arithmetic operator precedence

`**`	(highest precedence)
`*, /, //, %`	
`+, -`	(lowest precedence)

Operators of equal precedence are evaluated left to right with the exception of exponentiation (`**`), which is evaluated right to left (that is, "top down" when written using the conventional superscript notation). For example,

```
>>> 6 / 2 / 4         # the same as 3 / 4
0.75
>>> 6 / (2 / 4)       # the same as 6 / 0.5
12.0
>>> 2**2**3           # the same as 2**(2**3) == 2**8
256
>>> (2**2)**3         # the same as 4**3
64
```

In examples such as these, the text following the hash symbol, `#`, is a comment that is ignored by the interpreter. We shall sometimes use comments in this to explain more about a statement, but it is not necessary to type it in if you try out the code.

Methods and attributes of numbers

Python numbers are *objects* (in fact, everything in Python is an object) and have certain *attributes*, accessed using the "dot" notation: `<object>.<attribute>` (this use of the period has nothing to do with the decimal point appearing in a floating point number). Some attributes are simple values: for example, complex number objects have the attributes `real` and `imag` which are the real and imaginary (floating point) parts of the number:

```
>>> (4+5j).real
4.0
>>> (4+5j).imag
5.0
```

Other attributes are *methods*: callable functions that act on their object in some way.[4] For example, complex numbers have a method, `conjugate`, which returns the complex conjugate:

```
>>> (4+5j).conjugate()
(4-5j)
```

Here, the empty parentheses indicate that the method is to be *called*, that is, the function to calculate the complex conjugate is to be run on the number $4 + 5j$; if we omit them, as in `(4+5j).conjugate`, we are referring to the method itself (without calling it) – this method is itself an object!

[4] In this book, we will use the terms *method* and *function* interchangeably. In Python, everything is an object and the distinction is not as meaningful as it is in some other languages.

Integers and floating point numbers don't actually have very many attributes that it makes sense to use in this way, but if you're curious you can find out how many bits an integer takes up in memory by calling its `bit_length` method. For example,

```
>>> (3847298893721407).bit_length()
52
```

Note that Python allocates as much memory as is necessary to exactly represent the integer.

Mathematical functions

Two of the mathematical functions that are provided "by default" as so-called *built-ins* are `abs` and `round`.

`abs` returns the absolute value of a number as follows:

```
>>> abs(-5.2)
5.2
>>> abs(-2)
2
>>> abs(3+4j)
5.0
```

This is an example of *polymorphism*: the same function, `abs`, does different things to different objects. If passed a real number, x, it returns $|x|$, the non-negative magnitude of that number, without regard to sign; if passed a complex number, $z = x + iy$, it returns the modulus, $|z| = \sqrt{x^2 + y^2}$.

The `round` function (with one argument) rounds a floating point number to the nearest integer:

```
>>> round(-9.62)
-10
>>> round(7.5)
8
>>> round(4.5)
4
```

Note that in Python 3, this function employs *Banker's rounding*: if a number is mid way between two integers, then the even integer is returned.[5]

Python is a very modular language: functionality is available in packages and modules that are *imported* if they are needed but are not loaded by default: this keeps the memory required to run a Python program to a minimum and improves performance. For example, many useful mathematical functions are provided by the `math` module, which is imported with the statement

```
>>> import math
```

The `math` module concerns itself with floating point and integer operations (for functions of complex numbers, there is another module, called `cmath`). These are called

[5] In Python 2 the `round()` function rounds *away from zero* when two integers are equally close: thus `round(2.5)` is `3` but `round(-2.5)` is `-3`.

Table 2.3 Some functions provided by the `math` module. *Angular arguments are assumed to be in radians.*

`math.sqrt(x)`	$\sqrt{x}$
`math.exp(x)`	e^x
`math.log(x)`	$\ln x$
`math.log(x, b)`	$\log_b x$
`math.log10(x)`	$\log_{10} x$
`math.sin(x)`	$\sin(x)$
`math.cos(x)`	$\cos(x)$
`math.tan(x)`	$\tan(x)$
`math.asin(x)`	$\arcsin(x)$
`math.acos(x)`	$\arccos(x)$
`math.atan(x)`	$\arctan(x)$
`math.sinh(x)`	$\sinh(x)$
`math.cosh(x)`	$\cosh(x)$
`math.tanh(x)`	$\tanh(x)$
`math.asinh(x)`	arsinh(x)
`math.acosh(x)`	arcosh(x)
`math.atanh(x)`	artanh(x)
`math.hypot(x, y)`	The Euclidean norm, $\sqrt{x^2 + y^2}$
`math.factorial(x)`	$x!$
`math.erf(x)`	The error function at x
`math.gamma(x)`	The gamma function at x, $\Gamma(x)$
`math.degrees(x)`	Converts x from radians to degrees
`math.radians(x)`	Converts x from degrees to radians

by passing one (or sometimes more than one) number to them inside parentheses (the numbers are said to act as *arguments* to the function being called). For example,

```
>>> import math
>>> math.exp(-1.5)
0.22313016014842982
>>> math.cos(0)
1.0
>>> math.sqrt(16)
4.0
```

A complete list of the mathematical functions provided by the `math` module is available in the online documentation;[6] the more commonly used ones are listed in Table 2.3.

The `math` module also provides two very useful nonfunction attributes: `math.pi` and `math.e` give the values of π and e, the base of the natural logarithm, respectively.

It is possible to import the math module with '`from math import *`' and access its functions directly:

```
>>> from math import *
>>> cos(pi)
-1.0
```

[6] http://docs.python.org/3/library/math.html.

However, although this may be convenient for interacting with the Python shell, it is not recommended in Python programs. There is a danger of name conflicts (particularly if many modules are imported in this way), and makes it difficult to know which function comes from which module. Importing with `import math` keeps the functions bound to their module's *namespace*: thus, even though `math.cos` requires more typing it makes for code that is much easier to understand and maintain.

Example E2.2 As might be expected, mathematical functions can be strung together in a single expression:

```
>>> import math
>>> math.sin(math.pi/2)
1.0
>>> math.degrees(math.acos(math.sqrt(3)/2))
30.000000000000004
```

Note the finite precision here: the exact answer is $\arccos(\sqrt{3}/2) = 30°$.

The fact that the `int` function rounds down in casting a floating point number to an integer can be used to find the number of digits a positive integer has:

```
>>> int(math.log10(9999)) + 1
4
>>> int(math.log10(10000)) + 1
5
```

2.2.3 Variables

What is a variable?

When an object, such as a `float`, is created in a Python program or using the Python shell, memory is allocated for it: the location of this memory within the computer's architecture is called its *address*. The actual value of an object's address isn't actually very useful in Python, but if you're curious you can find it out by calling the `id` built-in method:

```
>>> id(20.1)
4297273888              # for example
```

This number refers to a specific location in memory that has been allocated to hold the `float` object with the value `20.1`.

For anything beyond the most basic usage, it is necessary to store the objects that are involved in a calculation or algorithm and to be able to refer to them by some convenient and meaningful name (rather than an address in memory). This is what *variables* are for.[7] A variable name can be assigned ("bound") to any object and used to identify that object in future calculations. For example,

---

[7] In Python, it is arguably better to talk of object *identifiers* or *identifier names* rather than variables, but we will not be too strict about this.

```
>>> a = 3
>>> b = -0.5
>>> a * b
-1.5
```

In this snippet, we create the `int` object with the value `3` and assign the variable name `a` to it. We then create the `float` object with the value `-0.5` and assign `b` to it. Finally, the calculation `a * b` is carried out: the values of `a` and `b` are multiplied together and the result returned. This result isn't assigned to any variable, so after being output to the screen it is thrown away. That is, the memory required to store the result, a `float` with the value `-1.5`, is allocated for long enough for it to be displayed to the user, but then it is gone.[8] If we need the result for some subsequent calculation, we should assign it to another variable:

```
>>> c = a * b
>>> c
-1.5
```

Note that we did not have to *declare* the variables before we assign them (tell Python that the variable name `a` is to refer to an integer, `b` is to refer to a floating point number, etc.), as is necessary in some computer languages. Python is a *dynamically typed* language and the necessary object type is inferred from its definition: in the absence of a decimal point, the number `3` is assumed to be an `int`; `-0.5` looks like a floating point number and so Python defines `b` to be a `float`.[9]

Variable names

There are some rules about what makes a valid variable name:

- Variable names are *case-sensitive*: `a` and `A` are different variables;
- Variable names can contain any letter, the underscore character ('_') and any digit (0–9) ...
- ... but must not *start with* a digit;
- A variable name must not be the same as one of the *reserved keywords* given in Table 2.4;
- The built-in constant names `True`, `False` and `None` cannot be assigned as variable names.

Most of the reserved keywords are pretty unlikely choices for variable names, with the exception of `lambda`. Python programmers often use `lam` if they need to use it. A good text editor will highlight the keywords as you type your program, so this is unlikely to cause confusion.

It is possible to give a variable the same name as a built-in function (e.g., `abs` and `round`), but that built-in function will no longer be available after such an assignment,

[8] Actually in an interactive Python session the result of the last calculation is stored in the special variable called _ (the underscore), so it isn't really thrown away until overwritten by the *next* calculation.

[9] This is sometimes called *duck-typing* after the phrase attributed to James Whitcomb Riley: "When I see a bird that walks like a duck and swims like a duck and quacks like a duck, I call that bird a duck."

Table 2.4 Python 3 reserved keywords

`and`	`assert`	`break`	`class`	`continue`
`def`	`del`	`elif`	`else`	`except`
`finally`	`for`	`from`	`global`	`if`
`import`	`in`	`is`	`lambda`	`nonlocal`
`not`	`or`	`pass`	`print`	`raise`
`return`	`try`	`while`	`yield`	

Note: in Python 2, `exec` is a keyword but `nonlocal` is not.

so this is probably best avoided – luckily, most have names that are unlikely to be chosen in practice.[10]

In addition to the rules mentioned earlier, there are certain style considerations that dictate good practice in naming variables:

- Variable names should be meaningful (`area` is better than `a`) ...
- ... but not too long (`the_area_of_the_triangle` is unwieldy);
- Generally, don't use `I` (uppercase `i`), `l` (lowercase `L`) or the uppercase letter `O`: they look too much like the digits 1 and 0;
- The variable names `i`, `j` and `k` are usually used as integer counters;
- Use lowercase names, with words separated by underscores rather than 'CamelCase': for example, `mean_height` and not `MeanHeight`.[11]

These and many other rules and conventions are codified in a style guide called PEP8 which forms part of the Python documentation[12] (see also Section 9.3.1).

Breaking these style rules will not result in your program failing to run, but it might make it harder to maintain and debug – the person you help might be yourself!

Example E2.3 *Heron's formula* gives the area, A, of a triangle with sides a, b, c as:

$$A = \sqrt{s(s-a)(s-b)(s-c)} \text{ where } s = \tfrac{1}{2}(a+b+c).$$

For example,

```
>>> a = 4.503
>>> b = 2.377
>>> c = 3.902
>>> s = (a + b + c) / 2
>>> area = math.sqrt(s * (s - a) * (s - b) * (s - c))   ❶
>>> area
4.63511081571606
```

❶ Don't forget to `import math` if you haven't already in this Python session.

[10] For a complete list of built-in function names, see http://docs.python.org/3/library/functions.html.

[11] CamelCase in Python is usually reserved for class names: see Section 4.6.2.

[12] http://legacy.python.org/dev/peps/pep-0008/.

Table 2.5 Python comparison operators

`==`	equal to
`!=`	not equal to
`>`	greater than
`<`	less than
`>=`	greater than or equal to
`<=`	less than or equal to

Example E2.4 The data type and memory address of the object referred to by a variable name can be found with the built-ins `type` and `id`:

```
>>> type(a)
<class 'float'>
>>> id(area)
4298539728       # for example
```

2.2.4 Comparisons and logic

Operators

The main comparison operators that are used in Python to compare objects (such as numbers) are given in Table 2.5.

The result of a comparison is a *boolean* object (of type `bool`) which has exactly one of two values: `True` or `False`. These are built-in constant keywords and cannot be reassigned to other values.[13] For example,

```
>>> 7 == 8
False
>>> 4 >= 3.14
True
```

Python is able, as far as possible without ambiguity, to compare objects of different types: the integer `4` is promoted to a `float` for comparison with the number `3.14`.

Note the importance of the difference between `==` and `=`. The single equals sign is an *assignment*, which does not return a value: the statement `a=7` assigns the variable `a` to the integer object `7` and that is all, whereas the expression `a==7` is a test: it returns `True` or `False` depending on the value of `a`.[14]

Care should be taken in comparing floating point numbers for equality. Because they are not stored exactly and calculations involving them frequently leads to a loss of precision, this can give unexpected results to the unwary. For example,

```
>>> a = 0.01
>>> b = 0.1**2
>>> a == b
False
```

[13] In Python 2, however, unhelpful assignments such as `True = False` were allowed.

[14] In some languages, such as C, assignment returns the value of whatever is being assigned, which can lead to some nasty and hard-to-find bugs when `=` is mistakenly used as a comparison operator.

In this example, 0.01 cannot be represented exactly as a floating point number but is (on my system) stored as a binary number equivalent to 0.01000000000000000208; the result of squaring the floating point representation of 0.1 on the other hand is 0.01000000000000000194, and these two numbers are not the same. See Section 9.1 for more information.

Logic operators

Comparisons can be modified and strung together with the logic operator keywords `and`, `not` and `or`. See Tables 2.6, 2.7 and 2.8. For example,

```
>>> 7.0 > 4 and -1 >= 0      # equivalent to True and False
False
>>> 5 < 4 or 1 != 2          # equivalent to False or True
True
```

In compound expressions such as these, the comparison operators are evaluated first, and then the logic operators in order of precedence: `not`, `and`, `or`. This precedence is overridden with parentheses, as for arithmetic. Thus,

```
>>> not 7.5 < 0.9 or 4 == 4
True
>>> not (7.5 < 0.9 or 4 == 4)
False
```

Table 2.6 Truth table for the `not` operator

P	not P
True	False
False	True

Table 2.7 Truth table for the `and` operator

P	Q	P and Q
True	True	True
False	True	False
True	False	False
False	False	False

Table 2.8 Truth table for the `or` operator

P	Q	P or Q
True	True	True
False	True	True
True	False	True
False	False	False

The truth tables for the logic operators are given below; note that, in common with most languages `or` in Python is the *inclusive or* variant for which `A or B` is `True` if both `A` and `B` are `True`, rather than the *exclusive or* operator (`A xor B` is `True` only if one but not both of `A` and `B` are `True`).

◊ Boolean equivalents and conditional assignment

In a logic test expression, it is not always necessary to make an explicit comparison to obtain a boolean value: Python will try to convert an object to a `bool` type if needed. For numeric objects, 0 evaluates to `False` and any nonzero value is `True`:

```
>>> a = 0
>>> a or 4 < 3       # same as: False or 4 < 3
False
>>>
>>> not a+1          # same as: not True
False
```

In this last example, addition has higher precedence than the logic operator `not`, so `a+1` is evaluated first to give `1`. This corresponds to boolean `True`, and so the whole expression is equivalent to `not True`. To explicitly convert an object to a boolean object, use the `bool` constructor:

```
>>> bool(-1)
True
>>> bool(0.0)
False
```

In fact, the `and` and `or` operators always return one of their operands and not just its `bool` equivalent. So, for example:

```
   >>> a = 0
❶  >>> a-2 or a
   -2
❷  >>> 4 > 3 and a-2
   -2
❸  >>> 4 > 3 and a
   0
```

Logic expressions are evaluated left to right, and those involving `and` or `or` are *short-circuited*: the second expression is only evaluated if necessary to decide the truth value of the whole expression. The three examples presented here can be analyzed as follows:
❶ In the first example, `a−2` is evaluated first: this is equal to −2, which is equivalent to `True`, so the `or` condition is fulfilled and the operand evaluating to `True` is returned immediately: −2.
❷ `4 > 3` is `True`, so the second expression must be evaluated to establish the truth of the `and` condition. `a−2` is equal to −2, which is also equivalent to `True`, so the `and` condition is fulfilled and −2 (as the most recently evaluated expression) is returned.
❸ In the last case, `a` is 0 which is equivalent to `False`: the `and` condition evaluates to `False` because of this, and so the return value is 0.

Python's special value, `None`

Python defines a single value, `None`, of the special type, `NoneType`. It is used to represent the absence of a defined value, for example, where no value is possible or relevant. This is particularly helpful in avoiding arbitrary default values (such as `0`, `1` or `−99`) for bad or missing data.

In a boolean comparison, `None` evaluates to `False`, but to test whether or not a variable, `x`, is equal to `None`, use

```
if x is None
```

and

```
if x is not None
```

rather than the shortcuts `if x` and `if not x`.[15]

Example E2.5 A common Python idiom is to assign a variable using the return value of a logic expression:

```
>>> a = 0
>>> b = a or -1
>>> b
-1
```

That is (for `a` understood to be an integer): "set `b` equal to the value of `a` unless `a==0`, in which case set `b` equal to -1."

2.2.5 Immutability and identity

The objects presented so far, such as integers and booleans, are *immutable*. Immutable objects do not change after they are created, though a variable name may be reassigned to refer to a different object from the one it was originally assigned to. For example, consider the assignments:

```
>>> a = 8
>>> b = a
```

The first line creates the integer object with value `8` in memory, and assigns the name `a` to it. The second line assigns the name `b` to the same object. You can see this by inspecting the address of the object refered to by each name:

```
>>> id(a)
4297273504
>>> id(b)
4297273504
```

Thus, `a` and `b` are references to the same integer object. Now suppose `a` is reassigned to a new number object:

---

[15] Recall that `not x` also evaluates to `True` if `x` is any of `0`, `False` or the empty string and so is not a very reliable way to test specifically if `x` is not set to `None`.

```
>>> a = 3.14
>>> a
3.14
>>> b
8
>>> id(a)
4298630152
>>> id(b)
4297273504
```

Note that the value of `b` has not changed: this variable still refers to the original `8`. The variable `a` now refers to a new, `float` object with the value `3.14` located at a new address. This is what is meant by immutability: it is not the "variable" that cannot change but the immutable object itself. This is illustrated in Figure 2.1.

A more convenient way to establish if two variables refer to the same object is to use the `is` operator, which determines object *identity*:

```
>>> a = 2432
>>> b = a
>>> a is b
True
>>> c = 2432
>>> c is a
False
>>> c == a
True
```

Here, the assignment `c = 2432` creates an entirely new integer object so `c is a` evaluates as `False`, even though `a` and `c` have the same *value*. That is, the two variables refer to different objects with the same value.

It is often necessary to change the value of a variable in some way, such as

```
>>> a = 800
>>> a = a + 1
>>> a
801
```

The integers `800` and `801` are immutable: the line `a = a + 1` creates a *new* integer object with the value `801` (the right-hand side is evaluated first) and assigns it to the

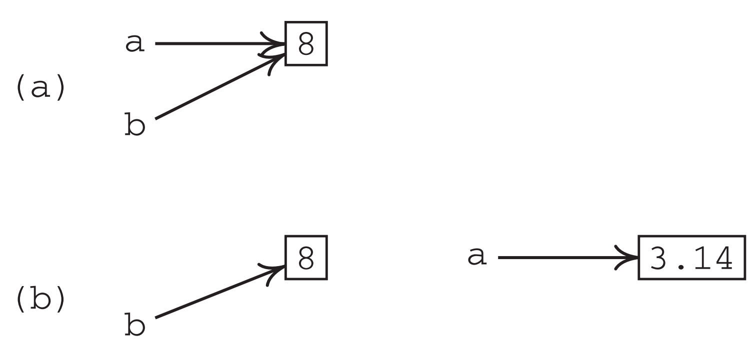


Figure 2.1 (a) Two variables referring to the same integer; (b) after reassigning the value of `a`.

variable name `a` (the old `800` is forgotten[16] unless some other variable refers to it). That is, `a` points to a different address before and after this statement.

This reassignment of a variable by an arithmetic operation on its value is so common that there is a useful shorthand notation: the *augmented assignment* `a += 5` is the same as `a = a + 5`. The operators `-=`, `*=`, `/=`, `//=`, `%=` work in the same way. C-style increment and decrement operations such as `a++` for `a += 1` are *not* supported in Python, however.[17]

Example E2.6 Python provides the operator `is not`: it is more natural to use `c is not a` than `not c is a`.

```
>>> a = 8
>>> b = a
>>> b is a
True
>>> b /= 2
>>> b is not a
True
```

◇ **Example E2.7** Given the previous discussion, it might come as a surprise to find that

```
>>> a = 256
>>> b = 256
>>> a is b
True
```

This happens because Python keeps a *cache* of commonly used, small integer objects (on my system, the numbers `–5–256`). To improve performance, the assignment `a = 256` attaches the variable name `a` to the existing integer object without having to allocate new memory for it. Because the same thing happens with `b`, the two variables in this case do, in fact, point to the same object. By contrast,

```
>>> a = 257
>>> b = 257
>>> a is b
False
```

2.2.6 Exercises

Questions

Q2.2.1 Predict the result of the following expressions and check them using the Python shell.

a. `2.7 / 2`
b. `2 / 4 - 1`

---

[16] That is, the memory assigned for it by Python is reclaimed ("garbage-collected") for general use.

[17] Assignment and augmented assignment in Python are statements not expressions and so do not return a value and cannot be chained together.

c. `2 // 4 - 1`
d. `(2 + 5) % 3`
e. `2 + 5 % 3`
f. `3 * 4 // 6`
g. `3 * (4 // 6)`
h. `3 * 2 ** 2`
i. `3 ** 2 * 2`

Q2.2.2 The operators listed in Table 2.1 are all *binary* operators: they take two operands (numbers) and return a single value. The symbol - is also used as a *unary* operator, which returns the negative value of the single operand on which it acts. For example,

```
>>> a = 4
>>> b = -a
>>> b
-4
```

Note that the expression `b = -a` (which sets the variable `b` to the negative value of `a`) is different from the expression `b -= a` (which subtracts `a` from `b` and stores the result in `b`). The unary – operator has a higher precedence than `*`, `/` and `%` but a lower precedence than exponentiation (`**`), so that, for example `-2 ** 4` is `-16` (i.e., $-(2^4)$, not $(-2)^4$).

Predict the result of the following expressions and check them using the Python shell.

a. `−2 ** 2`
b. `2 ** -2`
c. `−2 ** -2`
d. `2 ** 2 ** 3`
e. `2 ** 3 ** 2`
f. `−2 ** 3 ** 2`
g. `(−2) ** 3 ** 2`
h. `(−2) ** 2 ** 3`

Q2.2.3 Predict and explain the results of the following statements.

a. `9 + 6j / 2`
b. `complex(4, 5).conjugate().imag`
c. `complex(0, 3j)`
d. `round(2.5)`
e. `round(-2.5)`
f. `abs(complex(5, -4)) == math.hypot(4,5)`

Q2.2.4 Determine the value of i^i as a real number, where $i = \sqrt{-1}$.

Q2.2.5 Explain the (surprising?) behavior of the following short code:

```
>>> d = 8
>>> e = 2
>>> from math import *
```

```
>>> sqrt(d ** e)
16.88210319127114
```

Q2.2.6 Formally, the integer division, `a // b` is defined as the *floor* of a/b (sometimes written $\lfloor \frac{a}{b} \rfloor$) – that is, the largest integer less than or equal to `a / b`. The modulus or remainder, `a % b` (written a mod b), is then

$$a \bmod b = a - b\left\lfloor \frac{a}{b} \right\rfloor.$$

Use these definitions to predict the result of the following expressions and check them using the Python shell.

a. `7 // 4`
b. `7 % 4`
c. `-7 // 4`
d. `-7 % 4`
e. `7 // -4`
f. `7 % -4`
g. `-7 // -4`
h. `-7 % -4`

Q2.2.7 If two adjacent sides of a regular, six-sided die have the values a and b when viewed side-on and read left to right, the value on the top of the die is given by $3(a^3b - ab^3) \mod 7$.

Determine the value on the top of the die if (a) $a = 2, b = 6$, (b) $a = 3, b = 5$.

Q2.2.8 How many times must a sheet of paper (thickness, $t = 0.1$ mm but otherwise any size required) be folded to reach the moon (distance from Earth, $d = 384,400$ km)?

Q2.2.9 Predict the results of the following expressions and check them using the Python shell.

a. `not 1 < 2 or 4 > 2`
b. `not (1 < 2 or 4 > 2)`
c. `1 < 2 or 4 > 2`
d. `4 > 2 or 10/0 == 0`
e. `not 0 < 1`
f. `1 and 2`
g. `0 and 1`
h. `1 or 0`
i. `type(complex(2, 3).real) is int`

Q2.2.10 Explain why the following expression does not evaluate to 100.

```
>>> 10^2
8
```

Hint: refer to the Python documentation for *bitwise* operators.

Problems

P2.2.1 There is no exclusive-or operator provided "out of the box" by Python, but one can be constructed from the existing operators. Devise two different ways of doing this. The truth table for the xor operator is given in Table 2.9.

P2.2.2 Some fun with the `math` module:

a. What is special about the numbers $\sin\left(2017\sqrt[5]{2}\right)$ and $(\pi + 20)^i$?
b. What happens if you try to evaluate an expression, such as e^{1000}, which generates a number larger than the largest floating point number that can be represented in the default double precision? What if you restrict your calculation to integer arithmetic (e.g., by evaluating 1000!)?
c. What happens if you try to perform an undefined mathematical operation such as division by zero?
d. The maximum representable floating point number in IEEE 754 double precision is about 1.8×10^{308}. Calculate the length of the hypotenuse of a right angled triangle with opposite and adjacent sides 1.5×10^{200} and 3.5×10^{201} (i) using the `math.hypot()` function directly and (ii) without using this function.

P2.2.3 Some languages provide a `sign(a)` function which returns -1 if its argument, `a`, is negative and 1 otherwise. *Python does not provide such a function*, but the `math` module does include a function `math.copysign(x, y)`, which returns the absolute value of `x` with the sign of `y`. How would you use this function in the same way as the missing `sign(a)` function?

P2.2.4 The World Geodetic System is a set of international standards for describing the shape of the Earth. In the latest WGS-84 revision, the Earth's *geoid* is approximated to a reference ellipsoid that takes the form of an oblate spheroid with semi-major and semi-minor axes $a = 6378137.0$ m and $c = 6356752.314245$ m respectively.

Use the formula for the surface area of an oblate spheroid,

$$S_{\mathrm{obl}} = 2\pi a^2 \left(1 + \frac{1-e^2}{e}\mathrm{atanh}(e)\right), \quad \text{where } e^2 = 1 - \frac{c^2}{a^2},$$

to calculate the surface area of this reference ellipsoid and compare it with the surface area of the Earth assumed to be a sphere with radius 6371 km.

Table 2.9 Truth table for the `xor` operator

P	Q	P xor Q
True	True	False
False	True	True
True	False	True
False	False	False

2.3 Python objects I: strings

2.3.1 Defining a string object

A Python string object (of type `str`) is an ordered, immutable sequence of characters. To define a variable containing some constant text (a *string literal*), enclose the text in either single or double quotes:

```
>>> greeting = "Hello, Sir!"
>>> bye = 'À bientôt'
```

Strings can be concatenated using either the + operator or by placing them next to each other on the same line:

```
>>> 'abc' + 'def'
'abcdef'
>>> 'one ' 'two' ' three'
'one two three'
```

Python doesn't place any restriction on the length of a line, so a string literal can be defined in a single, quoted block of text. However, for ease of reading, it is usually a good idea to keep the lines of your program to a fixed maximum length (79 characters is recommended). To break up a string over two or more lines of code, use the line continuation character, '\' or (better) enclose the string literal in parentheses:

```
>>> long_string = 'We hold these truths to be self-evident,'\
...               ' that all men are created equal...'

>>> long_string = ('We hold these truths to be self-evident,'
...                ' that all men are created equal...')
```

This defines the variable `long_string` to hold a single line of text (with no carriage returns). The concatenation does not insert spaces so they need to be included explicitly if they are wanted. The spaces lining up the opening quotes in this example are optional but make the code easier to read.

If your string consists of a repetition of one or more characters, the `*` operator can be used to concatenate them the required number of times:

```
>>> 'a'*4
'aaaa'
>>> '-o-'*5
'-o--o--o--o--o-'
```

The *empty string* is defined simply as `s = ''` (two single quotes) or `s = ""`.

Finally, the built-in function, `str` converts an object passed as its argument into a string according to a set of rules defined by the object itself:

```
>>> str(42)
'42'
>>> str(3.4e5)
'340000.0'
>>> str(3.4e20)
'3.4e+20'
```

For finer control over the formatting of the string representation of numbers, see Section 2.3.7.

Example E2.8 Strings concatenated with the '+' operator can repeated with '*', but only if enclosed in parentheses:

```
>>> ('a'*4 + 'B')*3
'aaaaBaaaaBaaaaB'
```

2.3.2 Escape sequences

The choice of quotes for strings allows one to include the quote character itself inside a string literal – just define it using the other quote:

```
>>> verse = 'Quoth the Raven "Nevermore."'
```

But what if you need to include both quotes in a string? Or to include more than one line in the string? This case is handled by special *escape sequences* indicated by a backslash, \. The most commonly used escape sequences are listed in Table 2.10. For example,

```
>>> sentence = "He said, \"This parrot's dead.\""
❶ >>> sentence
'He said, "This parrot\'s dead."'
❷ >>> print(sentence)
He said, "This parrot's dead."
>>> subjects = 'Physics\nChemistry\nGeology\nBiology'
>>> subjects
'Physics\nChemistry\nGeology\nBiology'
>>> print(subjects)
Physics
Chemistry
Geology
Biology
```

❶ Note that just typing a variable's name at the Python shell prompt simply echoes its literal value back to you (in quotes).

Table 2.10 Common Python escape sequences

Escape sequence	Meaning
`\'`	Single quote (`'`)
`\"`	Double quote (`"`)
`\n`	Linefeed (LF)
`\r`	Carriage return (CR)
`\t`	Horizontal tab
`\b`	Backspace
`\\`	The backslash character itself
`\u`, `\U`, `\N{}`	Unicode character (see Section 2.3.3)
`\x`	Hex-encoded byte

❷ To produce the desired string including the proper interpretation of special characters, pass the variable to the `print` built-in function (see Section 2.3.6).

On the other hand, if you want to define a string to include character sequences such as '`\n`' *without them being escaped*, define a *raw string* prefixed with `r`:

```
>>> rawstring = r'The escape sequence for a new line is \n.'
>>> rawstring
'The escape sequence for a new line is \\n.'
>>> print(rawstring)
The escape sequence for a new line is \n.
```

When defining a block of text including several line endings it is often inconvenient to use `\n` repeatedly. This can be avoided by using *triple-quoted strings*: new lines defined within strings delimited by `"""` and `'''` are preserved in the string:[18]

```
a = """one
two
three"""
>>> print(a)
one
two
three
```

This is often used to create "docstrings" which document blocks of code in a program (see Section 2.7.1).

Example E2.9 The `\x` escape denotes a character encoded by the single-byte hex value given by the subsequent two characters. For example, the capital letter '`N`' has the value 78, which is `4e` in hex. Hence,

```
>>> '\x4e'
'N'
```

The *backspace* "character" is encoded as hex `08`, which is why `'\b'` is equivalent to `'\x08'`:

```
>>> 'hello\b\b\b\b\bgoodbye'
'hello\x08\x08\x08\x08\x08goodbye'
```

Sending this string to the `print()` function outputs the string formed by the sequence of characters in this string literal:

```
>>> print('hello\b\b\b\b\bgoodbye')
goodbye
```

2.3.3 Unicode

Python 3 strings are composed of *Unicode* characters. Unicode is a standard describing the representation of more than 100, 000 characters in just about every human language, as well as many other specialist characters such as scientific symbols. It does this by

[18] It is generally considered better to use three *double* quotes, `"""`, for this purpose.

assigning a number (*code point*) to every character; the numbers that make up a string are then encoded as a sequence of bytes.[19] For a long time, there was no agreed encoding standard, but the *UTF-8* encoding, which is used by Python 3 by default, has emerged as the most widely used today.[20] If your editor will not allow you to enter a character directly into a string literal, you can use its 16- or 32-bit hex value or its Unicode character name as an escape sequence:

```
>>> '\u00E9' # 16-bit hex value
'é'
>>> '\u000000E9' # 32-bit hex value
'é'
>>> '\N{LATIN SMALL LETTER E WITH ACUTE}' # by name
'é'
```

Example E2.10 Providing your editor or terminal allows it, and you can type them at your keyboard or paste them from elsewhere (e.g, a web browser or word processor), Unicode characters can be entered directly into string literals:

```
>>> creams = 'Crème fraîche, crème brûlée, crème pâtissière'
```

Python even supports Unicode variable names, so identifiers can use non-ASCII characters:

```
>>> Σ = 4
>>> crème = 'anglaise'
```

Needless to say, because of the potential difficulty in entering non-ASCII characters from a standard keyboard and because many distinct characters look very similar, this is not a good idea.

2.3.4 Indexing and slicing strings

Indexing (or "subscripting") a string returns a single character at a given location. Like all sequences in Python, strings are indexed with the first character having the index 0; this means that the final character in a string consisting of n characters is indexed at $n - 1$. For example,

```
>>> a = "Knight"
>>> a[0]
'K'
>>> a[3]
'g'
```

The character is returned in a `str` object of length 1. A non-negative index counts forward from the start of the string; there is a handy notation for the index of a string counting backward: a negative index, starting at -1 (for the final character) is used. So,

```
>>> a = "Knight"
>>> a[-1]
't'
```

[19] For a list of code points, see the official Unicode website's code charts at www.unicode.org/charts/.

[20] UTF-8 encoded Unicode encompasses the venerable 8-bit encoding of the ASCII character set (e.g., `A=65`).

```
>>> a[-4]
'i'
```

It is an error to attempt to index a string outside its length (here, with index greater than 5 or less than −6): Python raises an `IndexError`:

```
>>> a[6]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: string index out of range
```

Slicing a string, `s[i:j]`, produces a substring of a string between the characters at two indexes, *including* the first (`i`) but *excluding* the second (`j`). If the first index is omitted, 0 is assumed; if the second is omitted, the string is sliced to its end. For example,

```
>>> a = "Knight"
>>> a[1:3]
'ni'
>>> a[:3]
'Kni'
>>> a[3:]
'ght'
>>> a[:]
'Knight'
```

This can seem confusing at first, but it ensures that the length of a substring returned as `s[i:j]` has length `j-i` (for positive `i`, `j`) and that `s[:i] + s[i:] == s`. Unlike indexing, slicing a string outside its bounds does not raise an error:

```
>>> a= "Knight"
>>> a[3:10]
'ght'
>>> a[10:]
''
```

To test if a string contains a given substring, use the `in` operator:

```
>>> 'Kni' in 'Knight':
True
>>> 'kni' in 'Knight':
False
```

Example E2.11 Because of the nature of slicing, `s[m:n]`, `n-m` is always the length of the substring. In other words, to return `r` characters starting at index `m`, use `s[m:m+r]`. For example,

```
>>> s = 'whitechocolatespaceegg'
>>> s[:5]
'white'
>>> s[5:14]
'chocolate'
>>> s[14:19]
'space'
>>> s[19:]
'egg'
```

Example E2.12 The optional, third number in a slice specifies the *stride*. If omitted, the default is 1: return every character in the requested range. To return every *k*th letter, set the stride to `k`. Negative values of `k` reverse the string. For example,

```
>>> s = 'King Arthur'
>>> s[::2]
'Kn rhr'
>>> s[1::2]
'igAtu'
>>> s[-1:4:-1]
'ruhtrA'
```

This last slice can be explained as a selection of characters from the last (index `-1`) down to (but not including) character at index `4`, with stride `-1` (select every character, in the reverse direction).

A convenient way of reversing a string is to slice between default limits (by omitting the first and last indexes) with a stride of `-1`:

```
>>> s[::-1]
'ruhtrA gniK'
```

2.3.5 String methods

Python strings are *immutable* objects, and so it is not possible to change a string by assignment – for example, the following is an error:

```
>>> a = 'Knight'
>>> a[0] = 'k'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

New strings *can* be constructed from existing strings, but only as new objects. For example,

```
>>> a += ' Templar'
>>> print(a)
Knight Templar
>>> b = 'Black ' + a[:6]
>>> print(b)
Black Knight
```

To find the number of characters a string contains, use the `len` built-in method:

```
>>> a = 'Earth'
>>> len(a)
5
```

String objects come with a large number of methods for manipulating and transforming them. These are accessed using the usual dot notation we have met already – some of the more useful ones are listed in Table 2.11. In this and similar tables, text in *italics* is intended to be replaced by a specific value appropriate to the use of the method; italic text in `[`*`square brackets`*`]` denotes an optional argument.

Table 2.11 Some common string methods

Method	Description
`center(width)`	Return the string centered in a string with total number of characters *width*.
`endswith(suffix)`	Return `True` if the string ends with the substring *suffix*.
`startswith(prefix)`	Return `True` if the string starts with the substring *prefix*.
`index(substring)`	Return the lowest index in the string containing *substring*.
`lstrip([chars])`	Return a copy of the string with any of the leading characters specified by [*chars*] removed. If [*chars*] is omitted, any leading whitespace is removed.
`rstrip([chars])`	Return a copy of the string with any of the trailing characters specified by [*chars*] removed. If [*chars*] is omitted, any trailing whitespace is removed.
`strip([chars])`	Return a copy of the string with leading and trailing characters specified by [*chars*] removed. If [*chars*] is omitted, any leading and trailing whitespace is removed.
`upper()`	Return a copy of the string with all characters in uppercase.
`lower()`	Return a copy of the string with all characters in lowercase.
`title()`	Return a copy of the string with all words starting with capitals and other characters in lowercase.
`replace(old, new)`	Return a copy of the string with each substring *old* replaced with *new*.
`split([sep])`	Return a `list` (see Section 2.4.1) of substrings from the original string which are separated by the string *sep*. If *sep* is not specified, the separator is taken to be any amount of whitespace.
`join([list])`	Use the string as a separator in joining a `list` of strings.
`isalpha()`	Return `True` if all characters in the string are alphabetic and the string is not empty; otherwise return `False`.
`isdigit()`	Return `True` if all characters in the string are digits and the string is not empty; otherwise return `False`.

Because these methods each return a new string (remember that strings are immutable objects), they can be chained together:

```
>>> s = '-+-Python Wrangling for Beginners'
>>> s.lower().replace('wrangling', 'programming').lstrip('+-')
'python programming for beginners'
```

Example E2.13 Here are some possible manipulations using string methods:

```
>>> a = 'java python c++ fortran'
>>> a.isalpha()
False    ❶
>>> b = a.title()
>>> b
'Java Python C++ Fortran'
>>> c = b.replace(' ', '!\n')
>>> c
```

```
'Java!\nPython!\nC++!\nFortran'
>>> print(c)
Java!
Python!
C++!
Fortran!
>>> c.index('Python')
❷ 6
>>> c[6:].startswith('Py')
True
>>> c[6:12].isalpha()
True
```

❶ `a.isalpha()` is `False` because of the spaces and `'++'`.
❷ Note that `\n` is a single character.

2.3.6 The `print` function

One of the most obvious changes between Python 2 and Python 3 is in the way that `print` works. In the older version of Python, `print` was a *statement* that output the string representation of a list of objects, separated by spaces:

```
>>> ans = 6
>>> print 'Solve:', 2, 'x =', ans, 'for x'            # Python 2 only!
Solve: 2 x = 6 for x
```

(There was also a special syntax for redirecting the output to a file.) Python 3 adopts a more consistent and flexible approach: `print` is a built-in *function* (just like the others we have met such as `len` and `round`.) It takes a list of objects, and also, optionally, arguments `end` and `sep`, that specify which characters should end the string and which characters should be used to separate the printed objects respectively. Omitting these additional arguments gives the same result as the old `print` statement: the object fields are separated by a *single space* and the line is ended with a *newline* character.[21] For example,

```
>>> ans = 6
>>> print('Solve:', 2, 'x =', ans, 'for x')
Solve: 2 x = 6 for x
>>> print('Solve: ', 2, 'x = ', ans, ' for x', sep='', end='!\n')
Solve: 2x = 6 for x!
❶ >>> print()

>>> print('Answer: x =', ans/2)
Answer: x = 3.0
```

❶ Note that `print()` with no arguments just prints the default newline `end` character.

To suppress the newline at the end of a printed string, specify `end` to be the empty string: `end=''`:

[21] The specific newline character used depends on the operating system: for example, on a Mac it is '`\n`', on Windows it is two characters: '`\r\n`'.

```
>>> print('A line with no newline character', end='')
A line with no newline character>>>
```

The chevrons, `>>>`, at the end of this line form the prompt for the next Python command to be entered.

Example E2.14 `print` can be used to create simple text tables:

```
>>> heading = '| Index of Dutch Tulip Prices |'
>>> line = '+' + '-'*16 + '-'*13 + '+'
>>> print(line, heading, line,
...       '|    Nov 23 1636 |        100 |',
...       '|    Nov 25 1636 |        673 |',
...       '|    Feb  1 1637 |       1366 |', line, sep='\n')
...
+-----------------------------+
| Index of Dutch Tulip Prices |
+-----------------------------+
|    Nov 23 1636 |        100 |
|    Nov 25 1636 |        673 |
|    Feb  1 1637 |       1366 |
+-----------------------------+
```

2.3.7 String formatting

Introduction to Python 3 string formatting

In its simplest form, it is possible to use a string's `format` method to insert objects into it. The most basic syntax is

```
>>> '{} plus {} equals {}'.format(2, 3, 'five')
2 plus 3 equals five
```

Here, the `format` method is called on the string literal with the arguments `2`, `3` and `'five'` which are interpolated, in order, into the locations of the *replacement fields*, indicated by braces, `{}`. Replacement fields can also be numbered or named, which helps with longer strings and allows the same value to be interpolated more than once:[22]

```
>>> '{1} plus {0} equals {2}'.format(2, 3, 'five')
'3 plus 2 equals five'
>>> '{num1} plus {num2} equals {answer}'.format(num1=2, num2=3, answer='five')
'2 plus 3 equals five'
>>> '{0} plus {0} equals {1}'.format(2, 2+2)
'2 plus 2 equals 4'
```

Note that numbered fields can appear in any order and are indexed starting at 0.

Replacement fields can be given a minimum size within the string by the inclusion of an integer length after a colon as follows:

```
>>> '=== {0:12} ==='.format('Python')
'=== Python       ==='
```

---

[22] This type of string formatting was introduced into Python 2 as well, although only Python 2.7 supports *unnamed* replacement fields denoted by empty braces, `{}`.

If the string is too long for the minimum size, it will take up as many characters as needed (overriding the replacement field size specified):

```
>>> 'A number: <{0:2}>'.format(-20)
'A number: <-20>'       # -20 won't fit into 2 characters: 3 are used anyway
```

By default, the interpolated string is aligned to the left; this can be modified to align to the right or to center the string. The single characters `<`, `>` and `^` control the alignment:

```
>>> '=== {0:<12} ==='.format('Python')
'=== Python       ==='
>>> '=== {0:>12} ==='.format('Python')
'===       Python ==='
>>> '=== {0:^12} ==='.format('Python')
'===    Python    ==='
```

In these examples, the field is padded with spaces, but this fill character can also be specified. For example, to pad with hyphens in the last example, specify

```
>>> '=== {0:-^12} ==='.format('Python')
'=== ---Python--- ==='
```

It is even possible to pass the minimum field size as a parameter to be interpolated. Just replace the field size with a reference in braces as follows:

```
>>> a = 15
>>> 'This field has {0} characters: ==={1:>{2}}===.'.format(a, 'the field', a)
'This field has 15 characters: ===      the field===.'
```

Or with named interpolation:

```
>>> 'This field has {w} characters: ==={0:>{w}}===.'.format('the field', w=a)
'This field has 15 characters: ===      the field===.'
```

In each case, the second format specifier here has been taken to be `:>15`.

To insert the brace characters themselves into a formatted string, they must be doubled up: use '`{{`' and '`}}`'.

Formatting numbers

The Python 3 string `format` method provides a powerful way to format numbers.

The specifiers '`d`', '`b`', '`o`', '`x`'/'`X`' indicate a decimal, binary, octal and lowercase/uppercase hex *integer* respectively:

```
>>> a = 254
>>> 'a = {0:5d}'.format(a)      # decimal
'a =   254'
>>> 'a = {0:10b}'.format(a)     # binary
'a =   11111110'
>>> 'a = {0:5o}'.format(a)      # octal
'a =   364'
>>> 'a = {0:5x}'.format(a)      # hex (lowercase)
'a =    fe'
>>> 'a = {0:5X}'.format(a)      # hex (uppercase)
'a =    FE'
```

Numbers can be padded with zeros to fill out the specified field size by prefixing the minimum width with a `0`:

```
>>> a = 254
>>> 'a = {a:05d}'.format(a=a)
'a = 00254'
```

By default, the sign of a number is only output if it is negative. This behavior can also be customized by specifying, before the minimum width:

- '+': always output a sign;
- '-': only output a negative sign, the default; or
- ' ': output a leading space only if the number is positive.

This last option enables columns of positive and negative numbers to be lined up nicely:

```
>>> print('{0: 5d}\n{1: 5d}\n{2: 5d}'.format(-4510, 1001, -3026))
-4510
 1001
-3026
>>> a = -25
>>> b = 12
>>> s = '{0:+5d}\n{1:+5d}\n= {2:+3d}'.format(a, b, a+b)
>>> print(s)
  -25
  +12
= -13
```

There are also format specifiers for floating point numbers, which can be output to a chosen precision if desired. The most useful options are 'f': fixed-point notation, 'e'/'E': exponent (i.e., "scientific" notation), and 'g'/'G': a general format which uses scientific notation for very large and very small numbers.[23] The desired precision (number of decimal places) is specified as '.p' after the minimum field width. Some examples:

```
>>> a = 1.464e-10
>>> '{0:g}'.format(a)
'1.464e-10'
>>> '{0:10.2E}'.format(a)
'  1.46E-10'
>>> '{0:15.13f}'.format(a)
'0.0000000001464'
>>> '{0:10f}'.format(a)
'  0.000000'   ❶
```

❶ Note that Python will not protect you from this kind of rounding to zero if not enough space is provided for a fixed-point number.

Older C-style formatting

Python 3 also supports the less powerful, C-style format specifiers that are still in widespread use. In this formulation the replacement fields are specified with the minimum width and precision specifiers following a % sign. The objects whose values are to be interpolated are then given after the end of the string, following another % sign. They

[23] More specifically, the g/G specifier acts like f/F for numbers between 10^{-4} and 10^{p} where p is the desired precision (which defaults to 6), and acts like e/E otherwise.

must be enclosed in parentheses if there is more than one of them. The same letters for the different output types are used as earlier; strings must be specified explicitly with '%s'. For example,

```
>>> kB = 1.3806504-23
>>> 'Here\'s a number: %10.2e' % kB
"Here's a number:   1.38e-23"
>>> 'The same number formatted differently: %7.1e and %12.6e' % (kB, kB)
'The same number formatted differently: 1.4e-23 and 1.380650e-23'
>>> '%s is %g J/K' % ("Boltzmann's constant", kB)
"Boltzmann's constant is 1.38065e-23 J/K"
```

Example E2.15 Python can produce string representations of numbers for which thousands are separated by commas:

```
>>> '{:11,d}'.format(1000000)
'  1,000,000'
>>> '{:11,.1f}'.format(1000000.)
'1,000,000.0'
```

Here is another table, produced using several different string methods:

```
title = '|' + '{:^51}'.format('Cereal Yields (kg/ha)') + '|'
line = '+' + '-'*15 + '+' + ('-'*8 + '+')*4
row = '| {:<13} |' + ' {:6,d} |'*4
header = '| {:^13s} |'.format('Country') + (' {:^6d} |'*4).format(1980, 1990,
                                                                 2000, 2010)
print('+' + '-'*(len(title)-2) + '+',
      title,
      line,
      header,
      line,
      row.format('China', 2937, 4321, 4752, 5527),
      row.format('Germany', 4225, 5411, 6453, 6718),
      row.format('United States', 3772, 4755, 5854, 6988),
      line,
      sep='\n')

+---------------------------------------------------+
|               Cereal Yields (kg/ha)               |
+---------------+--------+--------+--------+--------+
|    Country    |  1980  |  1990  |  2000  |  2010  |
+---------------+--------+--------+--------+--------+
| China         |  2,937 |  4,321 |  4,752 |  5,527 |
| Germany       |  4,225 |  5,411 |  6,453 |  6,718 |
| United States |  3,772 |  4,755 |  5,854 |  6,988 |
+---------------+--------+--------+--------+--------+
```

2.3.8 Exercises

Questions

Q2.3.1 Slice the string `s='seehemewe'` to produce the following substrings:

a. `'see'`
b. `'he'`

c. `'me'`
d. `'we'`
e. `'hem'`
f. `'meh'`
g. `'wee'`

Q2.3.2 Write a single-line expression for determining if a string is a palindrome (reads the same forward as backward).

Q2.3.3 Predict the results of the following statements and check them using the Python shell.

```
>>> days = 'Sun Mon Tues Weds Thurs Fri Sat'
```

a. `print(days[days.index('M'):])`
b. `print(days[days.index('M'):days.index('Sa')].rstrip())`
c. `print(days[6:3:-1].lower()*3)`
d. `print(days.replace('rs', '').replace('s ', ' ')[::4])`
e. `print(' -*- '.join(days.split()))`

Q2.3.4 What is the output of the following code? How does it work?

```
>>> suff = 'thstndrdthththththth'
>>> n = 1
>>> print('{:d}{:s}'.format(n, suff[n*2:n*2+2]))
>>> n = 3
>>> print('{:d}{:s}'.format(n, suff[n*2:n*2+2]))
>>> n = 5
>>> print('{:d}{:s}'.format(n, suff[n*2:n*2+2]))
```

Q2.3.5 Consider the following (incorrect) tests to see if the string 's' has one of two values. Explain how these statements are interpreted by Python and give a correct alternative.

```
>>> s = 'eggs'
>>> s == ('eggs' or 'ham')
True

>>> s == ('ham' or 'eggs')
False
```

Problems

P2.3.1 a. Given a string representing a base-pair sequence (i.e., containing only the letters A, G, C and T), determine the fraction of G and C bases in the sequence. (*Hint:* strings have a `count` method, returning the number of occurrences of a substring.)

b. Using only string methods, devise a way to determine if a nucleotide sequence is a palindrome in the sense that it is equal to its own complementary sequence read backward. For example, the sequence TGGATCCA is palindromic because

its complement is ACCTAGGT which is the same as the original sequence backward. The complementary base pairs are (A, T) and (C, G).

P2.3.2 The table that follows gives the names, symbols, values, uncertainties and units of some physical constants.

Defining variables of the form

Name	Symbol	Value	Uncertainty	Units
Boltzmann constant	k_B	$1.3806504 \times 10^{-23}$	2.4×10^{-29}	$\mathrm{J\,K^{-1}}$
Speed of light	c	299792458	(def)	$\mathrm{m\,s^{-1}}$
Planck constant	h	$6.62606896 \times 10^{-34}$	3.3×10^{-41}	J s
Avogadro constant	N_A	$6.02214179 \times 10^{23}$	3×10^{16}	$\mathrm{mol^{-1}}$
Electron magnetic moment	μ_e	$-9.28476377 \times 10^{-24}$	2.3×10^{-31}	J/T
Gravitational constant	G	6.67428×10^{-11}	6.7×10^{-15}	$\mathrm{N\,m^2\,kg^{-2}}$

```
kB = 1.3806504e-23   # J/K
kB_unc = 2.4e-29     # uncertainty
kB_units = 'J/K'
```

use the string object's `format` method to produce the following output:

a.
```
kB = 1.381e-23 J/K
```

b.
```
G = 0.0000000000667428 Nm^2/kg^2
```

c. Using the same format specifier for each line,

```
kB   =  1.3807e-23 J/K
mu_e = -9.2848e-24 J/T
N_A  =  6.0221e+23 mol-1
c    =  2.9979e+08 m/s
```

d. Again, using the same format specifier for each line,

```
===    G = +6.67E-11  [Nm^2/kg^2]    ===
===   μe = -9.28E-24  [      J/T]    ===
```

Hint: the Unicode codepoint for the lowercase Greek letter mu is `U+03BC`.

e. (Harder). Produce the output below, in which the uncertainty (one standard deviation) in the value of each constant is expressed as a number in parentheses relative the preceding digits: that is, $6.62606896(33) \times 10^{-34}$ means $6.62606896 \times 10^{-34} \pm 3.3 \times 10^{-41}$.

```
G = 6.67428(67)e-11 Nm2/kg2
mu_e = -9.28476377(23)e-24 J/T
```

P2.3.3 Given the elements of a 3×3 matrix as the nine variables `a11`, `a12`, ... `a33`, produce a string representation of the matrix using formatting methods, (a) assuming the matrix elements are (possibly negative) real numbers to be given to one decimal place; (b) assuming the matrix is a permutation matrix with integer entries taking the values 0 or 1 only. For example,

```
>>> print(s_a)
[  0.0  3.4 -1.2 ]
[ -1.1  0.5 -0.2 ]
[  2.3 -1.4 -0.7 ]
>>> print(s_b)
[ 0 0 1 ]
[ 0 1 0 ]
[ 1 0 0 ]
```

P2.3.4 Find the Unicode code points for the planet symbols listed on the NASA website (http://solarsystem.nasa.gov/multimedia/display.cfm?IM_ID=167) which mostly fall within the hex range 2600–26FF: Miscellaneous Symbols (www.unicode.org/charts/PDF/U2600.pdf) and output a list of planet names and symbols.

2.4 Python objects II: lists, tuples and loops

2.4.1 Lists

Initializing and indexing lists

Python provides data structures for holding an ordered list of objects. In some other languages (e.g., C and Fortran) such a data structure is called an *array* and can hold only one type of data (e.g., an array of integers); the core array structures in Python, however, can hold a mixture of data types.

A Python *list* is an ordered, *mutable* array of objects. A list is constructed by specifying the objects, separated by commas, between square brackets, `[]`. For example,

```
>>> list1 = [1, 'two', 3.14, 0]
>>> list1
[1, 'two', 3.14, 0]
>>> a = 4
>>> list2 = [2, a, -0.1, list1, True]
>>> list2
[2, 4, -0.1, [1, 'two', 3.14, 0], True]
```

Note that a Python list can contain references to any type of object: strings, the various types of numbers, built-in constants such as the boolean value `True`, and even other lists. It is not necessary to declare the size of a list in advance of using it. An empty list can be created with `list0 = []`.

An item can be retrieved from the list by indexing it (remember Python indexes start at 0):

```
>>> list1[2]
3.14
>>> list2[-1]
True
>>> list2[3][1]
'two'
```

This last example retrieves the second (index: `1`) item of the fourth (index: `3`) item of `list2`. This is valid because the item `list2[3]` happens to be a list (the one also identified by the variable name `list1`), and `list1[1]` is the string `'two'`. In fact, since strings can also be indexed:

```
>>> list2[3][1][1]
'w'
```

To test for membership of a list, the operator `in` is used, as for strings:

```
>>> 1 in list1
True
>>> 'two' in list2:
False
```

This last expression evaluates to `False` because `list2` does not contain the string literal `'two'` even though it contains `list1` which does: the `in` operator does not recurse into lists-of-lists when it tests for membership.

Lists and mutability

Python lists are the first *mutable* object we have encountered. Unlike strings, which cannot be altered once defined, the items of a list can be reassigned:

```
>>> list1
[1, 'two', 3.14, 0]
>>> list1[2] = 2.72
>>> list1
[1, 'two', 2.72, 0]
>>> list2
[2, 4, -0.1, [1, 'two', 2.72, 0], True]
```

Note that not only has `list1` been changed, but `list2` (which contains `list1` as an item) *has also changed*.[24] This behavior catches a lot of people out to begin with, particularly if a list needs to be copied to a different variable.

```
>>> q1 = [1, 2, 3]
>>> q2 = q1
>>> q1[2] = 'oops'
>>> q1
[1, 2, 'oops']
>>> q2
[1, 2, 'oops']
```

Here, the variables `q1` and `q2` refer to the *same list*, stored in the same memory location, and because lists are mutable, the line `q1[2] = 'oops'` actually changes one of the stored values at that location; `q2` still points to the same location and so it appears to have changed as well. In fact, there is only one list (referred to by two variable names) and it is changed once. In contrast, integers are *immutable*, so the following does not change the value of `q[2]`:

```
>>> a = 3
>>> q = [1, 2, a]
>>> a = 4
>>> q
[1, 2, 3]
```

[24] Actually, it hasn't changed: it only ever contained a series of references to objects: the reference to `list1` is the same, even though the references within `list1` have changed.

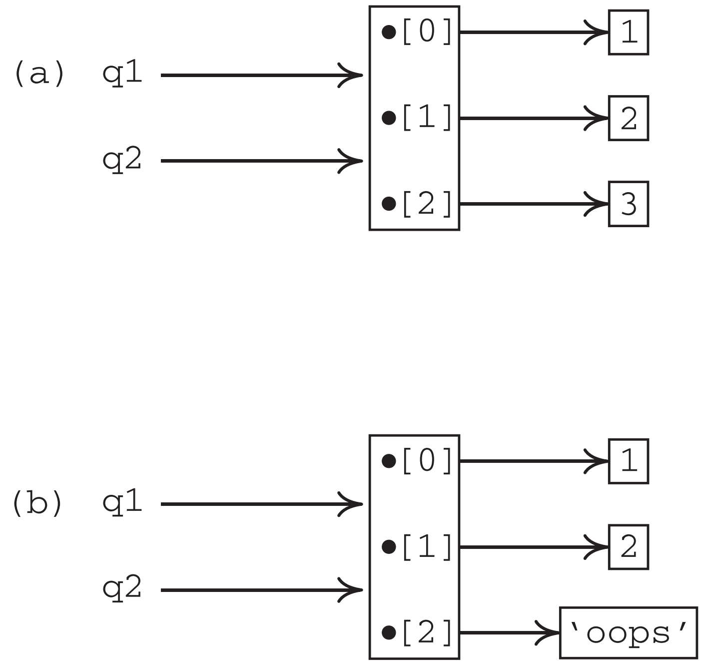


Figure 2.2 Two variables referring to the same list: (a) on initialization and (b) after setting `q1[2] = 'oops'`.

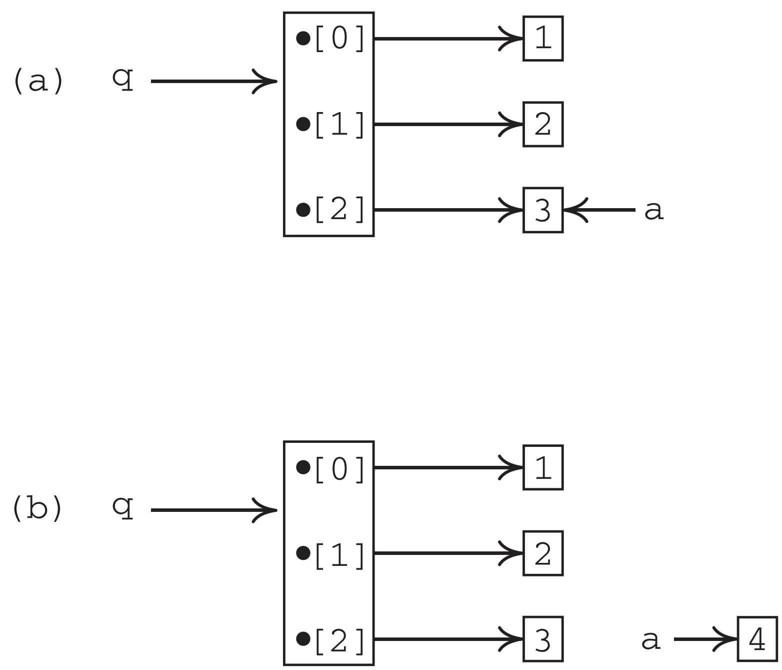


Figure 2.3 A list defined with `q = [1, 2, a]` where `a=3`: (a) on initialization and (b) after changing the value of a with `a=4`.

The assignment `a=4` creates a whole new integer object, quite independent of the original `3` that ended up in the list `q`. This original integer object isn't changed by the assignment (integers are immutable) and so the list is unchanged. This distinction is illustrated by Figures 2.2, 2.3 and 2.4.

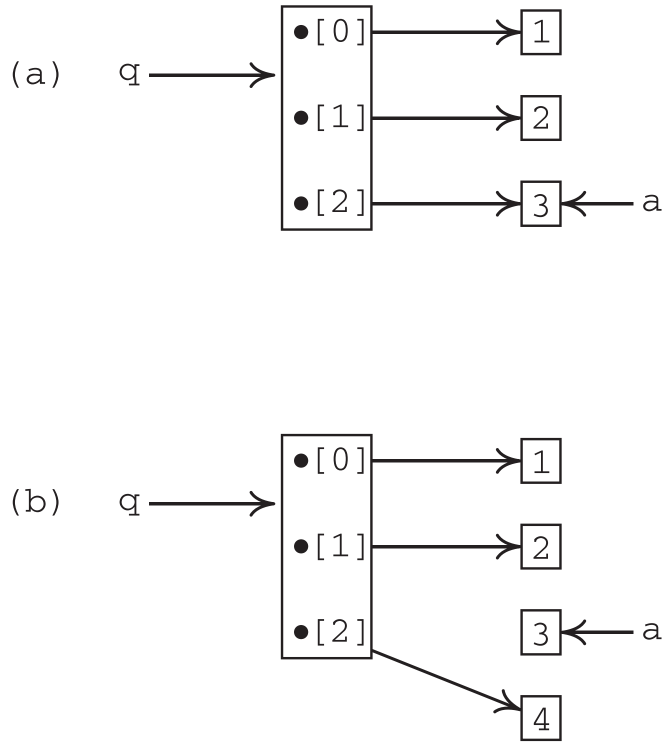


Figure 2.4 A list defined with `q = [1, 2, a]` where `a=3`: (a) on initialization and (b) after changing the value of q with `q[2]=4`.

Lists can be *sliced* in the same way as string sequences:

```
>>> q1 = [0., 0.1, 0.2, 0.3, 0.4, 0.5]
>>> q1[1:4]
[0.1, 0.2, 0.3]
>>> q1[::-1]        # return a reversed copy of the list
[0.5, 0.4, 0.3, 0.2, 0.1, 0.0]
>>> q1[1::2]        # striding: returns elements at 1, 3, 5
[0.1, 0.3, 0.5]
```

Taking a slice *copies the data* to a new list. Hence,

```
>>> q2 = q1[1:4]
>>> q2[1] = 99      # only affects q2
>>> q2
[0.1, 99, 0.3]
>>> q1
[0.0, 0.1, 0.2, 0.3, 0.4, 0.5]
```

List methods

Just as for strings, Python lists come with a large number of useful methods, summarized in Table 2.12. Because `list` objects are mutable, they can grow or shrink *in place*, that is, without having to copy the contents to a new object, as we had to do with strings. The relevant methods are

- `append`: add an item to the end of the list;

Table 2.12 Some common list methods

Method	Description
`append(element)`	Append *element* to the end of the list.
`extend(list2)`	Extend the list with the elements from *list2*.
`index(element)`	Return the lowest index of the list containing *element*.
`insert(index, element)`	Insert *element* at index *index*.
`pop()`	Remove and return the last element from the list.
`reverse()`	Reverse the list in place.
`remove(element)`	Remove the first occurrence of *element* from the list.
`sort()`	Sort the list in place.
`copy()`	Return a copy of the list.
`count(element)`	Return the number of elements equal to *element* in the list.

- `extend`: add one or more objects by copying them from another list;[25]
- `insert`: insert an item at a specified index and
- `remove`: remove a specified item from the list.

```
>>> q = []
>>> q.append(4)
>>> q
[4]
>>> q.extend([6, 7, 8])
>>> q
[4, 6, 7, 8]
>>> q.insert(1, 5)   # insert 5 at index 1
>>> q
[4, 5, 6, 7, 8]
>>> q.remove(7)
>>> q
[4, 5, 6, 8]
>>> q.index(8)
3   # the item 8 appears at index 3
```

Two useful list methods are `sort` and `reverse`, which sort and reverse the list *in place*. That is, they change the list object, but *do not return a value*:

```
>>> q = [2, 0, 4, 3, 1]
>>> q.sort()
>>> q
[0, 1, 2, 3, 4]
>>> q.reverse()
>>> q
[4, 3, 2, 1, 0]
```

If you do want a sorted *copy* of the list, leaving it unchanged, you can use the `sorted` built-in function:

[25] Actually, any Python object that forms a *sequence* that can be iterated over (e.g., a string) can be used as the argument to `extend`

```
>>> q = ['a', 'e', 'A', 'c', 'b']
>>> sorted(q)
['A', 'a', 'b', 'c', 'e']    # returns a new list
>>> q
['a', 'e', 'A', 'c', 'b']    # the old list is unchanged
```

By default, `sort()` and `sorted()` order the items in an array in *ascending order*. Set the optional argument `reverse=True` to return the items in descending order:

```
>>> q = [10, 5, 5, 2, 6, 1, 67]
>>> sorted(q, reverse=True)
[67, 10, 6, 5, 5, 2, 1]
```

Python 3, unlike Python 2, does not allow direct comparisons between strings and numbers, so it is an error to attempt to sort a list containing a mixture of such types:

```
>>> q = [5, '4', 2, 8]
>>> q.sort()
TypeError: unorderable types: str() < int()
```

Example E2.16 The methods `append` and `pop` make it very easy to use a `list` to implement the data structure known as a *stack*:

```
>>> stack = []
>>> stack.append(1)
>>> stack.append(2)
>>> stack.append(3)
>>> stack.append(4)
>>> print(stack)
[1, 2, 3, 4]
>>> stack.pop()
4
>>> print(stack)
[1, 2, 3]
```

The end of the list is the top of the stack from which items may be added or removed (think of a stack of dinner plates).

Example E2.17 The string method, `split` generates a list of substrings from a given string, split on a specified separator:

```
>>> s = 'Jan Feb Mar Apr May Jun'
>>> s.split()       # By default, splits on whitespace
['Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun']
>>> s = "J. M. Brown AND B. Mencken AND R. P. van't Rooden"
>>> s.split(' AND ')
['J. M. Brown', 'B. Mencken', "R. P. van't Rooden"]
```

2.4.2 Tuples

The `tuple` object

A `tuple` may be thought of as an immutable `list`. Tuples are constructed by placing the items inside parentheses:

```
>>> t = (1, 'two', 3.)
>>> t
(1, 'two', 3.0)
```

Tuples can be indexed and sliced in the same way as lists but, being immutable, they cannot be appended to, extended, or have elements removed from them:

```
>>> t = (1, 'two', 3.)
>>> t[1]
'two'
>>> t[2] = 4
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

Although a tuple itself is immutable, it may *contain* references to mutable objects such as lists. Hence,

```
>>> t = (1, ['a', 'b', 'd'], 0)
>>> t[1][2] = 'c'    # OK to change the list within the tuple
>>> t
(1, ['a', 'b', 'c'], 0)
```

An empty `tuple` is created with empty parentheses: `t0 = ()`. To create a `tuple` containing only one item (a *singleton*), however, it is not sufficient to enclose the item in parentheses (which could be confused with other syntactical use of parentheses); instead, the lone item is given a trailing comma: `t = ('one',)`.

Uses of `tuples`

In some circumstances, particularly for simple assignments such as those in the previous section, the parentheses around a tuple's items are not required:

```
>>> t = 1, 2, 3
>>> t
(1, 2, 3)
```

This usage is an example of *tuple packing*. The reverse, *tuple unpacking* is a common way of assigning multiple variables in one line:

```
>>> a, b, c = 97, 98, 99
>>> b
98
```

This method of assigning multiple variables is commonly used in preference to separate assignment statements either on different lines or (very un-Pythonically) on a single line, separated by semicolons:

```
a = 97; b = 98; c = 99      # Don't do this!
```

Tuples are useful where a sequence of items cannot or should not be altered. In the previous example, the `tuple` object only exists in order to assign the variables `a`, `b` and `c`. The values to be assigned: `97`, `98` and `99` are packed into a tuple for the purpose of this statement (to be unpacked into the variables), but once this has happened, the tuple object itself is destroyed. As another example, a function (Section 2.7) may return more

than one object: these objects are returned packed into a tuple. If you need any further persuading, `tuples` are slightly faster for many uses than `lists`.

Example E2.18 In an assignment using the '=' operator the right-hand side expression is evaluated first. This provides a convenient way to swap the values of two variables using tuples:

```
a, b = b, a
```

Here, the right-hand side is packed into a tuple object, which is then unpacked into the variables assigned on the left-hand side. This is more convenient than using a temporary variable:

```
t = a
a = b
b = t
```

2.4.3 Iterable objects

Examples of iterable objects

Strings, lists and tuples are all examples of data structures that are *iterable* objects: they are ordered sequences of items (characters in the case of strings, or arbitrary objects in the case of lists and tuples) which can be taken one at a time. One way of seeing this is to use the alternative method of initializing a list (or tuple) using the built-in constructor methods `list()` and `tuple()`. These take any iterable object and generate a list and a tuple respectively from its sequence of items. For example,

```
>>> list('hello')
['h', 'e', 'l', 'l', 'o']
>>> tuple([1, 'two', 3])
(1, 'two', 3)
```

Because the data elements are *copied* in the construction of a new object using these constructor methods, `list` is another way of creating an independent `list` object from another:

```
>>> a = [5, 4, 3, 2, 1]
>>> b = a          # b and a refer to the same list object
>>> b is a
True
>>> b = list(a)    # create an entirely new list object with the same contents as a
>>> b is a
False
```

Because slices also return a copy of the object references from a sequence, the idiom `b = a[:]` is often used in preference to `b = list(a)`.

`any` and `all`

The built-in function `any` tests whether any of the items in an iterable object are equivalent to `True`; `all` tests whether all of them are. For example,

```
>>> a = [1, 0, 0, 2, 3]
>>> any(a), all(a)
(True, False)         # some (but not all) of a's items are equivalent to True
>>> b = [[], False, 0.]
>>> any(b), all(b)
(False, False)        # none of b's items is equivalent to True
```

◊ *** syntax**

It is sometimes necessary to call a function with arguments taken from a list or other sequence. The `*` syntax, used in a function call unpacks such a sequence into positional arguments to the function (see also Section 2.7). For example, the `math.hypot` function takes two arguments, `a` and `b`, and returns the quantity $\sqrt{a^2 + b^2}$. If the arguments you wish to use are in a list or tuple, the following will fail:

```
>>> t = [3, 4]
>>> math.hypot(t)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: hypot expected 2 arguments, got 1
```

We tried to call `math.hypot()` with a single argument (the list object `t`), which is an error. We could index the list explicitly to retrieve the two values we need:

```
>>> t = [3, 4]
>>> math.hypot(t[0], t[1])
5.0
```

but a more elegant method is to *unpack* the list into arguments to the function with `*t`:

```
>>> math.hypot(*t)
5.0
```

`for` loops

It is often necessary to take the items in an iterable object one by one and do something with each in turn. Other languages, such as C, require this type of *loop* to refer to each item in turn by its integer index. In Python this is possible, but the more natural and convenient way is with the idiom:

`for` *`item`* `in` *`iterable object`*`:`

which yields each element of the iterable object in turn to be processed by the subsequent block of code. For example,

```
>>> fruit_list = ['apple', 'melon', 'banana', 'orange']
>>> for fruit in fruit_list:
...     print(fruit)
...
apple
melon
banana
orange
```

Each item in the `list` object `fruit_list` is taken in turn and assigned to the variable `fruit` for the block of statements following the '`:`' – each statement in this block must be indented by the same amount of whitespace. Any number of spaces or tab characters

could be used, but it is **strongly recommended to use four spaces** to indent code.[26] Loops can be nested – the inner loop block needs to be indented by the same amount of whitespace again as the outer loop (i.e. eight spaces):

```
>>> fruit_list = ['apple', 'melon', 'banana', 'orange']
>>> for fruit in fruit_list:
...     for letter in fruit:
...         print(letter, end='.')
...     print()
...
a.p.p.l.e.
m.e.l.o.n.
b.a.n.a.n.a.
o.r.a.n.g.e.
```

In this example, we iterate over the string items in `fruit_list` one by one, and for each string (fruit name), iterate over its letters. Each letter is printed followed by a full stop (the body of the inner loop). The last statement of the outer loop, `print()` forces a new line after each fruit.

Example E2.19 We have already briefly met the string method `join`, which takes a sequence of string objects and joins them together in a single string:

```
>>> ', '.join( ('one', 'two', 'three') )
'one, two, three'
>>> print('\n'.join(reversed(['one', 'two', 'three'])))
three
two
one
>>> ' '.join('hello')
'h e l l o'
```

Recall that strings are themselves iterable sequences, so the last statement joins the letters of `'hello'` with a single space.

The `range` type

Python provides an efficient method of referring to a sequence of numbers that forms a simple arithmetic progression: $a_n = a_0 + nd$ for $n = 0, 1, 2, \cdots$. In such a sequence, each term is spaced by a constant value, the *stride*, d. In the simplest case, one simply needs an integer counter which runs in steps of one from an initial value of zero: $0, 1, 2, \cdots, N - 1$. It would be possible to create a list to hold each of the values, but for most purposes this is wasteful of memory: it is easy to generate the next number in the sequence without having to store all of the numbers at the same time.

[26] The use of whitespace as part of the syntax of Python is one of its most contentious aspects. Some people used to languages such as C and Java which delimit code blocks with braces ({ . . . }) find it an anathema; others argue that code is almost always indented consistently to make it readable even when this isn't enforced by the grammar of the language and consider it less harmful.

Representing such arithmetic progressions for iterating over is the purpose of the `range` type. A `range` object can be constructed with up to three arguments defining the first integer, the integer to stop at and the stride (which can be negative).

```
range([a0=0], n, [stride=1])
```

The notation describing the `range` constructor here means that if the initial value, `a0`, is not given it is taken to be `0`; `stride` is also optional and if it is not given it is taken to be `1`. Some examples:

```
>>> a = range(5)          # 0,1,2,3,4
>>> b = range(1,6)        # 1,2,3,4,5
>>> c = range(0,6,2)      # 0,2,4
>>> d = range(10, 0, -2)  # 10,8,6,4,2
```

In Python 3, the object created by `range` *is not a list*.[27] Rather it is an iterable object that can produce integers on demand: `range` objects can be indexed, cast into lists and tuples, and iterated over:

```
>>> c[1]        # i.e. the second element of 0,2,4
2
>>> c[0]
0
>>> list(d)     # make a list from the range
[10, 8, 6, 4, 2]
>>> for x in range(5):
...     print(x)
0
1
2
3
4
```

Example E2.20 The *Fibonacci sequence* is the sequence of numbers generated by applying the rules:

$$a_1 = a_2 = 1, \quad a_i = a_{i-1} + a_{i-2}.$$

That is, the ith Fibonacci number is the sum of the previous two: $1, 1, 2, 3, 5, 8, 13, \cdots$.

We present two ways of generating the Fibonacci series. First, by appending to a list:

Listing 2.1 Calculating the Fibonacci series in a `list`

```
# eg2-i-fibonacci.py
# Calculates and stores the first n Fibonacci numbers

n = 100
fib = [1, 1]
for i in range(2, n+1):
    fib.append(fib[i-1] + fib[i-2])
print(fib)
```

[27] In Python 2, `range` returned a list and a second method, `xrange`, created the equivalent to Python 3's `range` object.

Alternatively, we can generate the series without storing more than two numbers at a time as follows:

Listing 2.2 Calculating the Fibonacci series without storing it

```
# eg2-ii-fibonacci.py
# Calculates the first n Fibonacci numbers

n = 100
# Keep track of the two most recent Fibonacci numbers
a, b = 1, 1
print(a, b, end='')
for i in range(2, n+1):
    # The next number (b) is a+b, and a becomes the previous b
    a, b = b, a+b
    print(' ', b, end='')
```

enumerate

Because `range` objects can be used to produce a sequence of integers, it is tempting to use them to provide the indexes of lists or tuples when iterating over them in a `for` loop:

```
>>> mammals = ['kangaroo', 'wombat', 'platypus']
>>> for i in range(len(mammals)):
        print(i, ':', mammals[i])
0 : kangaroo
1 : wombat
2 : platypus
```

This works, of course, but it is more natural to avoid the explicit construction of a `range` object (and the call to the `len` built-in) by using `enumerate`. This method takes an iterable object and produces, for each item in turn, a tuple `(count, item)`, consisting of a counting index and the item itself:

```
>>> mammals = ['kangaroo', 'wombat', 'platypus']
>>> for i, mammal in enumerate(mammals):
        print(i, ':', mammal)
0 : kangaroo
1 : wombat
2 : platypus
```

Note that each `(count, item)` tuple is unpacked in the `for` loop into the variables `i` and `mammal`. It is also possible to set the starting value of `count` to something other than 0 (although then it won't be the index of the item in the original list, of course):

```
>>> list(enumerate(mammals, 4))
[(4, 'kangaroo'), (5, 'wombat'), (6, 'platypus')]
```

◊ zip

What if you want to iterate over two (or more) sequences at the same time? This is what the `zip` built-in function is for: it creates an iterator object in which each item is a tuple of items taken in turn from the sequences passed to it:

```
>>> a = [1, 2, 3, 4]
>>> b = ['a', 'b', 'c', 'd']
>>> zip(a,b)
<builtins.zip at 0x104476998>
>>> for pair in zip(a,b):
...     print(pair)
...
(1, 'a')
(2, 'b')
(3, 'c')
(4, 'd')
>>> list(zip(a,b))      # convert to list
[(1, 'a'), (2, 'b'), (3, 'c'), (4, 'd')]
```

A nice feature of `zip` is that it can be used to *unzip* sequences of tuples as well:

```
>>> z = zip(a,b)        # zip
>>> A, B = zip(*z)      # unzip
>>> print(A, B)
(1, 2, 3, 4) ('a', 'b', 'c', 'd')
>>> list(A) == a, list(B) == b
(True, True)
```

`zip` does not copy the items into a new object, so it is memory-efficient and fast; but this means that you only get to iterate over the zipped items once and you can't index it:[28]

```
>>> z = zip(a, b):
>>> z[0]
TypeError: 'zip' object is not subscriptable

>>> for pair in z:
...     x = 0       # just some dummy operation performed on each iteration
...
>>> for pair in z:
...     print(pair)
...
        # (nothing: we've already exhausted the iterator z)
>>>
```

2.4.4 Exercises

Questions

Q2.4.1 Predict and explain the outcome of the following statements using the variables

```
s = 'hello'
a = [4, 10, 2]
```

a. `print(s, sep='-')`

[28] This is another difference between Python 2 and Python 3: in the older version of Python, `zip` returned a `list` of tuples.

b. `print(*s, sep='-')`

c. `print(a)`

d. `print(*a, sep='\thinspace\!\!')`

e. `list(range(*a))`

Q2.4.2 A `list` could be used as a simple representation of a polynomial, $P(x)$, with the items as the coefficients of the successive powers of x, and their indexes as the powers themselves. Thus, the polynomial $P(x) = 4 + 5x + 2x^3$ would be represented by the list `[4, 5, 0, 2]`. Why does the following attempt to differentiate a polynomial fail to produce the correct answer?

```
>>> P = [4, 5, 0, 2]
>>> dPdx = []
>>> for i, c in enumerate(P[1:]):
...     dPdx.append(i*c)
>>> dPdx
[0, 0, 4]              # wrong!
```

How can this code be fixed?

Q2.4.3 Given an ordered list of test scores, produce a list associating each score with a *rank* (starting with 1 for the highest score). Equal scores should have the same rank. For example, the input list `[87, 75, 75, 50, 32, 32]` should produce the list of rankings `[1,2,2,4,5,5]`.

Q2.4.4 Use a `for` loop to calculate π from the first 20 terms of the *Madhava series*:

$$\pi = \sqrt{12}\left(1 - \frac{1}{3\cdot 3} + \frac{1}{5\cdot 3^2} - \frac{1}{7\cdot 3^3} + \cdots\right).$$

Q2.4.5 For what iterable sequences, `x`, does the expression

`any(x) and not all(x)`

evaluate to `True`?

Q2.4.6 Explain why `zip(*z)` is the inverse of `z = zip(a, b)` – that is, while `z` pairs the items: `(a0, b0), (a1, b1), (a2, b2), ...`, `zip(*z)` separates them again: `(a0, a1, a2, ...), (b0, b1, b2, ...)`.

Q2.4.7 Sorting a list of tuples arranges them in order of the first element in each tuple first. If two or more tuples have the same first element, they are ordered by the second element, and so on:

```
>>> sorted([(3,1), (1,4), (3,0), (2, 2), (1, -1)])
[(1, -1), (1, 4), (2, 2), (3, 0), (3, 1)]
```

This suggests a way of using `zip` to sort one list using the elements of another. Implement this method on the data below to produce an ordered list of the average amount of sunshine in hours in London by month. Output the sunniest month first.

Jan	Feb	Mar	Apr	May	Jun
44.7	65.4	101.7	148.3	170.9	171.4
Jul	Aug	Sep	Oct	Nov	Dec
176.7	186.1	133.9	105.4	59.6	45.8

Problems

P2.4.1 Write a short Python program which, given an array of integers, `a`, calculates an array of the same length, `p`, in which `p[i]` is the product of all the integers in `a` *except* `a[i]`. So, for example, if `a = [1,2,3]`, then `p` is `[6, 3, 2]`.

P2.4.2 The *Hamming distance* between two equal-length strings is the number of positions at which the characters are different. Write a Python routine to calculate the Hamming distance between two strings, `s1` and `s2`.

P2.4.3 Using a tuple of strings naming the digits 0–9, create a Python program which outputs the representation of π as read aloud to 8 decimal places:

```
three point one four one five nine two six five
```

P2.4.4 Write a program to output a nicely formatted depiction of the first eight rows of Pascal's Triangle.

P2.4.5 A DNA sequence encodes each amino acid making up a protein as a three-nucleotide sequence called a *codon*. For example, the sequence fragment AGTCT-TATATCT contains the codons (AGT, CTT, ATA, TCT) if read from the first position (*"frame"*). If read in the second frame it yields the codons (GTC, TTA, TAT) and in the third (TCT, TAT, ATC).

Write some Python code to extract the codons into a list of 3-letter strings given a sequence and `frame` as an integer value (0, 1 or 2).

P2.4.6 The factorial function, $n! = 1 \cdot 2 \cdot 3 \cdot \cdots \cdot (n-1)n$ is the product of the first n positive integers and is provided by the `math` module's `factorial` method. The *double factorial* function, $n!!$, is the product of the positive *odd* integers up to and including n (which must itself be odd):

$$n!! = \prod_{i=1}^{(n+1)/2} (2i-1) = 1 \cdot 3 \cdot 5 \cdots (n-2) \cdot n.$$

Write a routine to calculate $n!!$ in Python.

As a bonus exercise, extend the formula to allow for even n as follows:

$$n!! = \prod_{i=1}^{n/2} (2i) = 2 \cdot 4 \cdot 6 \cdots (n-2) \cdot n.$$

P2.4.7 *Benford's Law* is an observation about the distribution of the frequencies of the first digits of the numbers in many different data sets. It is frequently found that the first

digits are not uniformly distributed, but follow the logarithmic distribution

$$P(d) = \log_{10}\left(\frac{d+1}{d}\right).$$

That is, numbers starting with 1 are more common than those starting with 2, and so on, with those starting with 9 the least common. The probabilities follow:

1	0.301
2	0.176
3	0.125
4	0.097
5	0.079
6	0.067
7	0.058
8	0.051
9	0.046

Benford's Law is most accurate for data sets which span several orders of magnitude, and can be proved to be exact for some infinite sequences of numbers.

1. Demonstrate that the first digits of the first 500 Fibonacci numbers (see Example E2.20) follow Benford's Law quite closely.
2. The length of the amino acid sequences of 500 randomly chosen proteins are provided in the file `ex2-4_e_ii_protein_lengths.py` which can be downloaded from scipython.com/ex/aba. This file contains a list, `naa`, which can be imported at the start of your program with

   ```
   from ex2-4_e_ii_protein_lengths import naa
   ```

   To what extent does the distribution of protein lengths obey Benford's Law?

2.5 Control flow

Few computer programs are executed in a purely linear fashion, one statement after another as written in the source code. It is more likely that during the program execution, data objects are inspected and blocks of code executed conditionally on the basis of some test carried out on them. Thus, all practical languages have the equivalent of an *if-then-(else)* construction. This section explains the syntax of Python's version of this clause and covers a further kind of loop: the `while` loop.

2.5.1 `if ... elif ... else`

The `if ... elif ... else` construction allows statements to be executed conditionally, depending on the result of one or more logical tests (which evaluate to the boolean values `True` or `False`):

```
if <logical expression 1>:
    <statements 1>
elif <logical expression 2>:
    <statements 2>
...
else:
    <statements>
```

That is, if `<logical expression 1>` evaluates to `True`, `<statements 1>` are executed; otherwise, if `<logical expression 2>` evaluates to `True`, `<statements 2>` are executed, and so on; if none of the preceding logical expressions evaluate to `True`, the statements in the block of code following `else:` are executed. These statement blocks are indented with whitespace, as for the `for` loop. For example,

```
for x in range(10):
    if x <= 3:
        print(x, 'is less than or equal to three')
    elif x > 5:
        print(x, 'is greater than five')
    else:
        print(x, 'must be four or five, then')
```

produces the output:

```
0 is less than or equal to three
1 is less than or equal to three
2 is less than or equal to three
3 is less than or equal to three
4 must be four or five, then
5 must be four or five, then
6 is greater than five
7 is greater than five
8 is greater than five
9 is greater than five
```

It is not necessary to enclose test expressions such as `x <= 3` in parentheses, as it is in C, for example, but the colon following the test is mandatory. The test expressions don't, in fact, have to evaluate explicitly to the boolean values `True` and `False`: as we have seen, other data types are taken to be equivalent to `True` unless they are `0` (`int`) or `0.` (`float`), the empty string, `''`, empty list, `[]`, the empty tuple, `()`, and so forth or Python's special type, `None` (see Section 2.2.4). Consider:

```
for x in range(10):
    if x % 2:
        print(x, 'is odd!')
    else:
        print(x, 'is even!')
```

This works because `x % 2 = 1` for odd integers, which is equivalent to `True` and `x % 2 = 0` for even integers, which is equivalent to `False`.

There is **no** `switch ... case ... finally` construction in Python – equivalent control flow can be achieved with `if ... elif ... endif` or with *dictionaries* (see Section 4.2).

Example E2.21 In the Gregorian calendar a year is a *leap year* if it is divisible by 4 with the exceptions that years divisible by 100 are *not* leap years unless they are also divisible by 400. The following Python program determines if `year` is a leap year.

Listing 2.3 Determining if a year is a leap year

```
year = 1900

if not year % 400:
    is_leap_year = True
elif not year % 100:
    is_leap_year = False
elif not year % 4:
    is_leap_year = True
else:
    is_leap_year = False

s_ly = 'is a' if is_leap_year else 'is not a'
print('{:4d} {:s} leap year'.format(year, s_ly))
```

Hence the output:

```
1900 is not a leap year
```

2.5.2 `while` loops

Whereas a `for` loop is established for a fixed number of iterations, statements within the block of a `while` loop execute only and as long as some condition holds:

```
>>> i = 0
>>> while i < 10:
...     i += 1
...     print(i, end='.')
...
>>> print()
1.2.3.4.5.6.7.8.9.10.
```

The counter `i` is initialized to `0`, which is less than 10 so the `while` loop begins. On each iteration, `i` is incremented by one and its value printed. When `i` reaches `10`, on the following iteration `i < 10` is `False`: the loop ends and execution continues after the loop, where `print()` outputs a newline.

Example E2.22 A more interesting example of the use of a `while` loop is given by this implementation of Euclid's algorithm for finding the greatest common divisor of two numbers, $\gcd(a, b)$:

```
>>> a, b = 1071, 462
>>> while b:
...    a, b = b, a % b
...
>>> print(a)
21
```

The loop continues until `b` divides `a` exactly; on each iteration, `b` is set to the remainder of `a//b` and then `a` is set to the old value of `b`. Recall that the integer 0 evaluates as boolean `False` so `while b:` is equivalent to `while b != 0:`.

2.5.3 More control flow: `break`, `pass`, `continue` and `else`

`break`

Python provides three further statements for controlling the flow of a program. The `break` command, issued inside a loop, immediately ends that loop and moves execution to the statements following the loop:

```
x = 0
while True:
    x += 1
    if not (x % 15 or x % 25):
        break
print(x, 'is divisible by both 15 and 25')
```

The `while` loop condition here is (literally) always `True` so the only escape from the loop occurs when the `break` statement is reached. This occurs only when the counter `x` is divisible by both `15` and `25`. The output is therefore:

```
75 is divisible by both 15 and 25
```

Similarly, to find the index of the first occurrence of a negative number in a list:

```
alist = [0, 4, 5, -2, 5, 10]
for i, a in enumerate(alist):
    if a < 0:
        break
print(a, 'occurs at index', i)
```

Note that after escaping from the loop, the variables `i` and `a` have the values that they had within the loop at the `break` statement.

`continue`

The `continue` statement acts in a similar way to `break` but instead of breaking out of the containing loop, it immediately forces the next iteration of the loop without completing the statement block for the current iteration. For example,

```
for i in range(1, 11):
    if i % 2:
        continue
    print(i, 'is even!')
```

prints only the even integers $2, 4, 6, 8, 10$: if `i` is not divisible by 2 (and hence `i % 2` is 1, equivalent to `True`), that loop iteration is canceled and the loop resumed with the next value of `i` (the `print` statement is skipped).

`pass`

The `pass` command does nothing. It is useful as a “stub” for code that has not yet been written but where a statement is syntactically required by Python’s whitespace convention.

```
>>> for i in range(1, 11):
...     if i == 6:
...         pass    # do something special if i is 6
...     if not i % 3:
...        print(i, 'is divisible by 3')
...
3 is divisible by 3
6 is divisible by 3
9 is divisible by 3
```

If the `pass` statement had been `continue` the line `6 is divisible by 3` would not have been printed: execution would have returned to the top of the loop and `i=7` instead of continuing to the second `if` statement.

◊ `else`

A `for` or `while` loop may be followed by an `else` block of statements, which will be executed only if the loop finished "normally" (that is, *without* the intervention of a `break`). For `for` loops, this means these statements will be executed after the loop has reached the end of the sequence it is iterating over; for `while` loops, they are executed when the `while` condition becomes `False`. For example, consider again our program to find the first occurrence of a negative number in a list. This code behaves rather oddly if there aren't any negative numbers in the list:

```
>>> alist = [0, 4, 5, 2, 5, 10]
>>> for i, a in enumerate(alist):
...     if a < 0:
...         break
...
>>> print(a, 'occurs at index', i)
10 occurs at index 5
```

It outputs the index and number of the last item in the list (whether it is negative or not). A way to improve this is to notice when the `for` loop runs through every item without encountering a negative number (and hence the `break`) and output a message:

```
>>> alist = [0, 4, 5, 2, 5, 10]
... for i, a in enumerate(alist):
...     if a < 0:
...         print(a, 'occurs at index', i)
...         break
... else:
...    print('no negative numbers in the list')
...

no negative numbers in the list
```

As another example, consider this (not particularly elegant) routine for finding the largest factor of a number `a > 2`:

```
a = 1013
b = a - 1
while b != 1:
    if not a % b:
```

```
            print('the largest factor of', a, 'is', b)
            break
        b -= 1
else:
    print(a, 'is prime!')
```

`b` is the largest factor not equal to `a`. The `while` loop continues as long as `b` is not equal to `1` (in which case `a` is prime) and decrements `b` after testing if `b` divides `a` exactly; if it does, `b` is the highest factor of `a`, and we `break` out of the `while` loop.

Example E2.23 A simple "turtle" virtual robot lives on an infinite two-dimensional plane on which its location is always an integer pair of (x, y) coordinates. It can face only in directions parallel to the x and y axes (i.e. 'North,' 'East,' 'South' or 'West') and it understands four commands:

- `F`: move forward one unit;
- `L`: turn left (counterclockwise) by 90°;
- `R`: turn right (clockwise) by 90°;
- `S`: stop and exit.

The following Python program takes a list of such commands as a string and tracks the turtle's location. The turtle starts at $(0, 0)$, facing in the direction $(1, 0)$ ('East'). The program ignores (but warns about) invalid commands and reports when the turtle crosses its own path.

Listing 2.4 A virtual turtle robot

```
# eg2-turtle.py
commands = 'FFFFFLFFFLFFFFRRRFXFFFFFFS'

# Current location, current facing direction
x, y = 0, 0
dx, dy = 1, 0
# Keep track of the turtle's location in the list of tuples, locs
locs = [(0, 0)]

❶ for cmd in commands:
    if cmd == 'S':
        # Stop command
        break
    if cmd == 'F':
        # Move forward in the current direction
        x += dx
        y += dy
        if (x, y) in locs:
            print('Path crosses itself at: ({}, {})'.format(x,y))
        locs.append((x,y))
        continue
    if cmd in 'LR':
        # Turn to the left (counterclockwise) or right (clockwise)
        # L => (dx, dy): (1,0) -> (0, 1) -> (-1,0) -> (0,-1) -> (1,0)
        # R => (dx, dy): (1,0) -> (0,-1) -> (-1,0) -> (0, 1) -> (1,0)
        sgn = 1
```

```
            if dy != 0:
                sgn = -1
            if cmd == 'R':
                sgn = -sgn
            dx, dy = sgn * dy, sgn * dx
            continue
        # if we're here it's because we don't recognize the command: warn
        print('Unknown command:', cmd)
❷ else:
        # We exhausted the commands without encountering an S for STOP
        print('Instructions ended without a STOP')

    # Plot a path of asterisks
    # First find the total range of x and y values encountered
❸ x, y = zip(*locs)
    xmin, xmax = min(x), max(x)
    ymin, ymax = min(y), max(y)
    # The grid size needed for the plot is (nx, ny)
    nx = xmax - xmin + 1
    ny = ymax - ymin + 1
    # Reverse the y-axis so that it decreases *down* the screen
    for iy in reversed(range(ny)):
        for ix in range(nx):
            if (ix+xmin, iy+ymin) in locs:
                print('*', end='')
            else:
                print(' ', end='')
        print()
```

❶ We can iterate over the string `commands` to take its characters one at a time.
❷ Note that the `else:` clause to the `for` loop is only executed if we do not `break` out of it on encountering a STOP command.
❸ We unzip the list of tuples, `locs`, into separate sequences of the *x* and *y* coordinates with `zip(*locs)`.

The output produced from the commands given is:

```
Unknown command: X
Path crosses itself at: (1, 0)
 *****
 *   *
 *   *
******
 *
 *
 *
 *
```

2.5.4 Exercises

Questions

Q2.5.1 Write a Python program to normalize a list of numbers, `a`, such that its values lie between 0 and 1. Thus, for example, the list `a = [2,4,10,6,8,4]` becomes `[0.0, 0.25, 1.0, 0.5, 0.75, 0.25]`.

Hint: use the built-ins `min`, and `max` which return the minimum and maximum values in a sequence respectively; for example, `min(a)` returns `2` in the earlier mentioned list.

Q2.5.2 Write a `while` loop to calculate the *arithmetic-geometric mean* (AGM) of two positive real numbers, x and y, defined as the limit of the sequences:

$$a_{n+1} = \frac{1}{2}(a_n + b_n)$$
$$b_{n+1} = \sqrt{a_n b_n},$$

starting with $a_0 = x$, $b_0 = y$. Both sequences converge to the same number, denoted $\mathrm{agm}(x, y)$. Use your loop to determine *Gauss's constant*, $G = 1/\mathrm{agm}(1, \sqrt{2})$.

Q2.5.3 The game of "Fizzbuzz" involves counting, but replacing numbers divisible by 3 with the word '*Fizz*,' those divisible by 5 with '*Buzz*,' and those divisible by both 3 and 5 with '*FizzBuzz*.' Write a program to play this game, counting up to 100.

Q2.5.4 Straight-chain alkanes are hydrocarbons with the general stoichiometric formula C_nH_{2n+2}, in which the carbon atoms form a simple chain: for example, butane, C_4H_{10} has the structural formula that may be depicted $H_3CCH_2CH_2CH_3$. Write a program to output the structural formula of such an alkane, given its stoichiometry (assume $n > 1$). For example, given `stoich='C8H18'`, the output should be

```
H3C-CH2-CH2-CH2-CH2-CH2-CH2-CH3
```

Problems

P2.5.1 Modify your solution to Problem P2.4.4 to output the first 50 rows of Pascal's triangle, but instead of the numbers themselves, output an asterisk if the number is odd and a space if it is even.

P2.5.2 The *iterative weak acid* approximation determines the hydrogen ion concentration, $[H^+]$ of an acid solution from the acid dissociation constant, K_a, and the acid concentration, c, by successive application of the formula

$$[H^+]_{n+1} = \sqrt{K_a \left(c - [H^+]_n\right)},$$

starting with $[H^+]_0 = 0$. The iterations are continued until $[H^+]$ changes by less than some predetermined, small tolerance value.

Use this method to determine the hydrogen ion concentration, and hence the pH ($= -\log_{10}[H^+]$) of a $c = 0.01$ M solution of acetic acid ($K_a = 1.78 \times 10^{-5}$). Use the tolerance `TOL = 1.e-10`.

P2.5.3 The *Luhn algorithm* is a simple checksum formula used to validate credit card and bank account numbers. It is designed to prevent common errors in transcribing the number, and detects all single-digit errors and almost all transpositions of two adjacent digits. The algorithm may be written as the following steps:

1. Reverse the number.
2. Treating the number as an array of digits, take the even-indexed digits (where the indexes *start at 1*) and double their values. If a doubled digit results in a number greater than 10, add the two digits (e.g., the digit 6 becomes 12 and hence 1 + 2 = 3).
3. Sum this modified array.
4. If the sum of the array modulo 10 is 0 the credit card number is valid.

Write a Python program to take a credit card number as a string of digits (possibly in groups, separated by spaces) and establish if it is valid or not. For example, the string `'4799 2739 8713 6272'` is a valid credit card number, but any number with a single digit in this string changed is not.

P2.5.4 *Hero's method* for calculating the square root of a number, S, is as follows: starting with an initial guess, x_0, the sequence of numbers $x_{n+1} = \frac{1}{2}(x_n + S/x_n)$ are successively better approximations to $\sqrt{S}$. Implement this algorithm to estimate the square root of 2117519.73 to two decimal places and compare with the "exact" answer provided by the `math.sqrt` method. For the purpose of this exercise, start with an initial guess, $x_0 = 2000$.

P2.5.5 Write a program to determine the tomorrow's date given a string representing today's date, `today`, as either "`D/M/Y`" or "`M/D/Y`." Cater for both British and US-style dates when parsing `today` according to the value of a boolean variable `us_date_style`. For example, when `us_date_style` is `False` and `today` is `'3/4/2014'`, tomorrow's date should be reported as `'4/4/2014'`.[29] (*Hint:* use the algorithm for determining if a year is a leap year, which is provided in the example to Section 2.5.1.)

P2.5.6 Write a Python program to determine $f(n)$, the number of trailing zeros in $n!$, using the special case of *de Polignac's formula*:

$$f(n) = \sum_{i=1} \left\lfloor \frac{n}{5^i} \right\rfloor,$$

where $\lfloor x \rfloor$ denotes the *floor* of x, the largest integer less than or equal to x.

P2.5.7 The *hailstone sequence* starting at an integer $n > 0$ is generated by the repeated application of the three rules:

- if $n = 1$, the sequence ends;
- if n is even, the next number in the sequence is $n/2$;
- if n is odd, the next number in the sequence is $3n + 1$.

a. Write a program to calculate the hailstone sequence starting at 27.
b. Let the *stopping time* be the number of numbers in a given hailstone sequence. Modify your hailstone program to return the stopping time instead of the numbers

[29] In practice, it would be better to use Python's `datetime` library (described in Section 4.5.3), but avoid it for this exercise.

themselves. Adapt your program to demonstrate that the hailstone sequences started with $1 \le n \le 100$ agree with the *Collatz conjecture* (that all hailstone sequences stop eventually).

P2.5.8 The algorithm known as the *Sieve of Eratosthenes* finds the prime numbers in a list $2, 3, \cdots, n$. It may be summarized as follows, starting at $p = 2$, the first prime number:

Step 1. Mark all the multiples of p in the list as nonprime (that is, the numbers mp where $m = 2, 3, 4, \cdots$: these numbers are *composite*.

Step 2. Find the first unmarked number greater than p in the list. If there is no such number, stop.

Step 3. Let p equal this new number and return to Step 1.

When the algorithm stops, the unmarked numbers are the primes.

Implement the Sieve of Eratosthenes in a Python program and find all the primes under 10000.

P2.5.9 *Euler's totient function*, $\phi(n)$, counts the number of positive integers less than or equal to n that are relatively prime to n. (Two numbers, a and b, are relatively prime if the only positive integer that divides both of them is 1; that is, if $\gcd(a, b) = 1$.)

Write a Python program to compute $\phi(n)$ for $1 \le n < 100$.

(*Hint:* you could use Euclid's algorithm for the greatest common divisor given in the example to Section 2.5.2.)

P2.5.10 The value of π may be approximated by Monte Carlo methods. Consider region of the xy-plane bounded by $0 \le x \le 1$ and $0 \le y \le 1$. By selecting a large number of random points within this region and counting the proportion of them lying beneath the function $y = \sqrt{1 - x^2}$ describing a quarter-circle, one can estimate $\pi/4$, this being the area bounded by the axes and $y(x)$. Write a program to estimate the value of π by this method.

Hint: use Python's `random` module. The method `random.random()` generates a (pseudo-)random number between 0. and 1. See Section 4.5.1 for more information.

P2.5.11 Write a program to take a string of text (words, perhaps with punctuation, separated by spaces) and output the same text with the middle letters shuffled randomly. Keep any punctuation at the end of words. For example, the string:

Four score and seven years ago our fathers brought forth on this continent a new nation, conceived in liberty, and dedicated to the proposition that all men are created equal.

might be rendered:

Four sorce and seevn yeras ago our fhtaers bhrogut ftroh on this cnnoientt a new noitan, cvieecond in lbrteiy, and ddicetead to the ptosoiporin that all men are cetaerd euaql.

Hint: `random.shuffle` shuffles a *list* of items in place. See Section 4.5.1.

P2.5.12 The *electron configuration* of an atom is the specification of the distribution of its electrons in atomic orbitals. An atomic orbital is identified by a *principal quantum number*, $n = 1, 2, 3, \cdots$ defining a *shell* comprised of one or more *subshells* defined

by the *azimuthal quantum number*, $l = 0, 1, 2, \cdots, n-1$. The values $l = 0, 1, 2, 3$ are refered to be the letters s, p, d and f respectively. Thus, the first few orbitals are 1s ($n = 1, l = 0$), 2s ($n = 2, l = 0$), 2p ($n = 2, l = 1$), 3s ($n = 3, l = 0$), and each shell has n subshells. A maximum of $2(2l + 1)$ electrons may occupy a given subshell.

According to the *Madelung rule*, the N electrons of an atom fill the orbitals in order of increasing $n + l$ such that whenever two orbitals have the same value of $n + l$, they are filled in order of increasing n. For example, the ground state of Titanium ($N = 22$) is predicted (and found) to be $1s^2 2s^2 2p^6 3s^2 3p^6 4s^2 3d^2$.

Write a program to predict the electronic configurations of the elements up to Rutherfordium ($N = 104$). The output for Titanium should be

```
Ti: 1s2.2s2.2p6.3s2.3p6.4s2.3d2
```

A Python list containing the element symbols in order may be downloaded from scipython.com/ex/abb.

As a bonus exercise, modify your program to output the configurations using the convention that the part of the configuration corresponding to the outermost *closed shell*, a noble gas configuration, is replaced by the noble gas symbol in square brackets; thus,

```
Ti: [Ar].4s2.3d2
```

the configuration of Argon being `1s2.2s2.2p6.3s2.3p6`.

2.6 File input/output

Until now, data has been hard-coded into our Python programs, and output has been to the console (the terminal). Of course, it will frequently be necessary to input data from an external file and to write data to an output file. To acheive this, Python has `file` objects.

2.6.1 Opening and closing a file

A `file` object is created by opening a file with a given *filename* and *mode*. The filename may be given as an absolute path, or as a path relative to the directory in which the program is being executed. `mode` is a string with one of the values given in Table 2.13. For example, to open a file for text-mode writing:

```
>>> f = open('myfile.txt', 'w')
```

`file` objects are closed with the `close` method: for example, `f.close()`. Python closes any open file objects automatically when a program terminates.

2.6.2 Writing to a file

The `write` method of a `file` object writes a *string* to the file and returns the number of characters written:

```
>>> f.write('Hello World!')
12
```

Table 2.13 File modes

mode argument	Open mode
r	text, read-only (the default)
w	text, write (an existing file with the same name will be overwritten)
a	text, append to an existing file
r+	text, reading and writing
rb	binary, read-only
wb	binary, write (an existing file with the same name will be overwritten)
ab	binary, append to an existing file
rb+	binary, reading and writing

More helpfully, the `print` built-in takes an argument, `file`, to specify where to redirect its output :

```
>>> print(35, 'Cl', 2, sep='', file=f)
```

writes '35Cl2' to the file opened as `file` object `f` instead of to the console.

Example E2.24 The following program writes the first four powers of the numbers between 1 and 1,000 in comma-separated fields to the file `powers.txt`:

```
f = open('powers.txt', 'w')
for i in range(1,1001):
    print(i, i**2, i**3, i**4, sep=', ', file=f)
f.close()
```

The file contents are

```
1, 1, 1, 1
2, 4, 8, 16
3, 9, 27, 81
...
999, 998001, 997002999, 996005996001
1000, 1000000, 1000000000, 1000000000000
```

2.6.3 Reading from a file

To read `n` bytes from a file, call `f.read(n)`. If `n` is omitted, the entire file is read in.[30]

`readline()` reads a single line from the file, up to and including the newline character. The next call to `readline()` reads in the next line, and so on. Both `read()` and `readline()` return an empty string when they reach the end of the file.

To read all of the lines into a list of strings in one go, use `f.readlines()`.

`file` objects are iterable, and looping over a (text) `file` returns its lines one at a time:

[30] To quote the official documentation: "it's your problem if the file is twice as large as your machine's memory."

```
>>> for line in f:
...     print(line, end='')   ❶
...
First line
Second line
...
```

❶ Because `line` retains its newline character when read in, we use `end=''` to prevent `print` from adding another, which would be output as a blank line.

You probably want to use this method if your file is very large unless you really do want to store every line in memory. See Section 4.3.4 concerning Python's `with` statement for more best practice in file handling.

Example E2.25 To read in the numbers from the file `powers.txt` generated in the previous example, the columns must be converted to lists of integers. To do this, each line must be split into its fields and each field explicitly converted to an `int`:

```
f = open('powers.txt', 'r')
squares, cubes, fourths = [], [], []
for line in f.readlines():
    fields = line.split(',')
    squares.append(int(fields[1]))
    cubes.append(int(fields[2]))
    fourths.append(int(fields[3]))
f.close()
n = 500
print(n, 'cubed is', cubes[n-1])
```

The output is

```
500 cubed is 125000000
```

In practice, it is better to use `numpy` (see Chapter 6) to read in data files such as these.

2.6.4 Exercises

Problems

P2.6.1 The coast redwood tree species, *Sequoia sempervirens*, includes some of the oldest and tallest living organisms on Earth. Some details concerning individual trees are given in the tab-delimited text file `redwood-data.txt`, available at scipython.com/ex/abd. (Data courtesy of the Gymnosperm database, www.conifers.org/cu/Sequoia.php)

Write a Python program to read in this data and report the tallest tree and the tree with the greatest diameter.

P2.6.2 Write a program to read in a text file and censor any words in it that are on a list of banned words by replacing their letters with the same number of asterisks. Your program should store the banned words in lowercase but censor examples of these words in any case. Assume there is no punctuation.

Table 2.14 Parameters used in the definition of ESI

i	Parameter	Earth value, $x_{i,\oplus}$	Weight, w_i
1	Radius	1.0	0.57
2	Density	1.0	1.07
3	Escape velocity, v_{esc}	1.0	0.7
4	Surface temperature	288 K	5.58

As a bonus exercise, handle text that contains punctuation. For example, given the list of banned words: `['C', 'Perl', 'Fortran']` the sentence

```
'Some alternative programming languages to Python are C, C++, Perl, Fortran and Java.'
```

becomes

```
'Some alternative programming languages to Python are *, C++, ****, ******* and Java.'
```

P2.6.3 The *Earth Similarity Index* (ESI) attempts to quantify the physical similarity between an astronomical body (usually a planet or moon) and Earth. It is defined by

$$\mathrm{ESI}_j = \prod_{i=1}^{n} \left(1 - \left|\frac{x_{i,j} - x_{i,\oplus}}{x_{i,j} + x_{i,\oplus}}\right|\right)^{w_i/n}$$

where the parameters $x_{i,j}$ are described, and their terrestrial values, $x_{i,\oplus}$ and weights, w_i given in Table 2.14. The radius, density and escape velocities are taken *relative to* the terrestrial values. The ESI lies between 0 and 1, with the values closer to 1 indicating closer similarity to Earth (which has an ESI of exactly 1: Earth is identical to itself!).

The file `ex2-6-g-esi-data.txt` available from scipython.com/ex/abc contains the earlier mentioned parameters for a range of astronomical bodies. Use these data to calculate the ESI for each of the bodies. Which has properties "closest" to those of the Earth?

P2.6.4 Write a program to read in a two-dimensional array of strings into a list of lists from a file in which the string elements are separated by one or more spaces. The number of rows, m, and columns, n, may not be known in advance of opening the file. For example, the text file

```
A B C D
E F G H
I J K L
```

should create an object, `grid`, as

```
[['A', 'B', 'C', 'D'], ['E', 'F', 'G', 'H'], ['I', 'J', 'K', 'L']]
```

Read like this, `grid` contains a list of the array's *rows*. Once the array has been read in, write loops to output the *columns* of the array:

```
[['A', 'E', 'I'], ['B', 'F', 'J'], ['C', 'G', 'K'], ['D', 'H', 'L']]
```

Harder: also output all its diagonals read in one direction:

```
[['A'], ['B', 'E'], ['C', 'F', 'I'], ['D', 'G', 'J'], ['H', 'K'], ['L']]
```

and the other direction:

```
[['D'], ['C', 'H'], ['B', 'G', 'L'], ['A', 'F', 'K'], ['E', 'J'], ['I']]
```

2.7 Functions

A Python *function* is a set of statements that are grouped together and named so that they can be run more than once in a program. There are two main advantages to using functions. First, they enable code to be reused without having to be replicated in different parts of the program; second, they enable complex tasks to be broken into separate procedures, each implemented by its own function – it is often much easier and more maintainable to code each procedure individually than to code the entire task at once.

2.7.1 Defining and calling functions

The def statement defines a function, gives it a name, and lists the arguments (if any) that the function expects to receive when called. The function's statements are written in an indented block following this def. If at any point during the execution of this statement block a return statement is encountered, the specified values are returned to the caller. For example,

```
❶ >>> def square(x):
   ...     x_squared = x**2
   ...     return x_squared
   ...
   >>> number = 2
❷ >>> number_squared = square(number)
   >>> print(number, 'squared is', number_squared)
   2 squared is 4
❸ >>> print('8 squared is', square(8))
   8 squared is 64
```

❶ The simple function named square takes a single argument, x. It calculates x**2 and returns this value to the caller. Once defined, it can be called any number of times. ❷ In the first example, the return value is assigned to the variable number_squared; ❸ in the second example, it is fed straight into the print method for output to the console.

To return two or more values from a function, pack them into a tuple. For example, the following program defines a function to return both roots of the quadratic equation $ax^2 + bx + c$ (assuming it *has* two real roots):

```
import math

def roots(a, b, c):
    d = b**2 - 4*a*c
    r1 = (-b + math.sqrt(d)) / 2 / a
    r2 = (-b - math.sqrt(d)) / 2 / a
    return r1, r2

print(roots(1., -1., -6.))
```

When run, this program outputs, as expected:

```
(3.0, -2.0)
```

It is not necessary for a function to explicitly return any object: functions that fall off the end of their indented block without encountering a `return` statement return Python's special value, `None`.

Function definitions can appear anywhere in a Python program, but a function cannot be referenced before it is defined. Functions can even be *nested*, but a function defined inside another is not (directly) accessible from outside that function.

Docstrings

A function *docstring* is a string literal that occurs as the first statement of the function definition. It should be written as a triple-quoted string on a single line if the function is simple, or on multiple lines with an initial one-line summary for more detailed descriptions of complex functions. For example,

```
def roots(a, b, c):
    """Return the roots of ax^2 + bx + c."""
    d = b**2 - 4*a*c
    ...
```

The docstring becomes the special `__doc__` attribute of the function:

```
>>> roots.__doc__
'Return the roots of ax^2 + bx + c.'
```

A docstring should provide details about *how to use the function*: which arguments to pass it and which objects it returns,[31] but should not generally include details of the specific *implementation* of algorithms used by the function (these are best explained in *comments*, preceded by `#`).

Docstrings are also used to provide documentation for classes and modules (see Sections 4.5 and 4.6.2).

Example E2.26 In Python, *functions are "first class" objects*: they can have variable identifiers assigned to them, they can be passed as arguments to other functions, and they can even be returned *from* other functions. A function is given a name when it is defined, but that name can be reassigned to refer to a different object (don't do this unless you mean to!) if desired.

As the following example demonstrates, it is possible for more than one variable name to be assigned to the same function object.

```
>>> def cosec(x):
...     """Return the cosecant of x, cosec(x) = 1/sin(x)."""
...     return 1./math.sin(x)
...
>>> cosec
<function cosec at 0x100375170>
```

[31] For larger projects, docstrings document an application programming interface (API) for the project.

```
>>> cosec(math.pi/4)
1.4142135623730951
>>> csc = cosec
>>> csc
<function cosec at 0x100375170>
>>> csc(math.pi/4)
1.4142135623730951
```

❶ The assignment `csc = cosec` associates the identifier (variable name) `csc` with the same function object as the identifier `cosec`: this function can then be called with `csc()` as well as with `cosec()`.

2.7.2 Default and keyword arguments

Keyword arguments

In the previous example, the arguments have been passed to the function in the order in which they are given in the function's definition (these are called *positional* arguments). It is also possible to pass the arguments in an arbitrary order by setting them explicitly as *keyword arguments*:

```
roots(a=1., c=-6., b=-1.)
roots(b=-1., a=1., c=-6.)
```

If you mix nonkeyword (positional) and keyword arguments the former must come first; otherwise Python won't know to which variable the positional argument corresponds:

```
>>> roots(1., c=6., b=-1.)  # OK
(3.0, -2.0)
>>> roots(b=-1., 1., -6.)   # Oops: which is a and which is c?
  File "<stdin>", line 1
SyntaxError: non-keyword arg after keyword arg
```

Default arguments

Sometimes you want to define a function that takes an *optional* argument: if the caller doesn't provide a value for this argument, a default value is used. Default arguments are set in the function definition:

```
>>> def report_length(value, units='m'):
...     return 'The length is {:.2f} {}'.format(value, units)
>>> report_length(33.136, 'ft')
'The length is 33.14 ft'
>>> report_length(10.1)
'The length is 10.10 m'
```

Default arguments are assigned *when the Python interpreter first encounters the function definition*. This can lead to some unexpected results, particularly for mutable arguments. For example,

```
>>> def func(alist = []):
...     alist.append(7)
...     return alist
...
```

```
>>> func()
[7]
>>> func()
[7, 7]
>>> func()
[7, 7, 7]
```

The default argument to the function `func` here is an empty list, but it is the specific empty list assigned when the function is defined. Therefore, each time `func` is called this specific list grows.

Example E2.27 Default argument values are assigned *when the function is defined*. Therefore, if a function is defined with an argument defaulting to the value of some variable, subsequently changing that variable *will not change the default*:

```
>>> default_units = 'm'
>>> def report_length(value, units=default_units):
...     return 'The length is {:.2f} {}'.format(value, units)
...
>>> report_length(10.1)
'The length is 10.10 m'
>>> default_units = 'cubits'
>>> report_length(10.1)
'The length is 10.10 m'
```

The default units used by the function `report_length` are unchanged by the reassignment of the variable name `default_units`: the default value is set to the string object referred to by `default_units` when the `def` statement is encountered by the Python compiler (`'m'`) and cannot be changed subsequently.

2.7.3 Scope

A function can define and use its own variables. When it does so, those variables are *local* to that function: they are not available outside the function. Conversely, variables assigned outside all function `def`s are *global* and are available everywhere within the program file. For example,

```
>>> def func():
...     a = 5
...     print(a,b)
...
>>> b = 6
>>> func()
5 6
```

The function `func` defines a variable `a`, but prints out both `a` and `b`. Because the variable `b` isn't defined in the local scope of the function, Python looks in the global scope, where it finds `b = 6`, so that is what is printed. It doesn't matter that `b` hasn't been defined when the function is *defined*, but of course it must be before the function is *called*.

What happens if a function defines a variable with the same name as a global variable? In this case, within the function the local scope is searched first when resolving variable

names, so it is the object pointed to by the local variable name that is retrieved. For example,

```
>>> def func():
...     a = 5
...     print(a)
...
>>> a = 6
>>> func()
5
>>> print(a)
6
```

Note that the local variable `a` exists only within the body of the function; it just happens to have the same name as the global variable `a`. It disappears after the function exits and it doesn't overwrite the global `a`.

Python's rules for resolving scope can be summarized as "LEGB": first *local* scope, then *enclosing* scope (for nested functions), then *global* scope, and finally *built-ins*–if you happen to give a variable the same name as a built-in function (such as `range` or `len`), then that name resolves to your variable (in local or global scope) and not to the original built-in. It is therefore generally not a good idea to name your variables after built-ins.

◊ The `global` and `nonlocal` keywords

We have seen that it is possible to access variables defined in scopes other than the local function's. Is it possible to *modify* them ("*rebind*" them to new objects)? Consider the distinction between the behavior of the following functions:

```
>>> def func1():
...     print(x)    # OK, providing x is defined in global or enclosing scope
...
>>> def func2():
...     x += 1      # Not OK: can't modify x if it isn't local
...
>>> x = 4
>>> func1()
4
>>> func2()
UnboundLocalError: local variable 'x' referenced before assignment
```

If you really do want to change variables that are defined outside the local scope, you must first declare within the function body that this is your intention with the keywords `global` (for variables in global scope) and `nonlocal` (for variables in enclosing scope, for example, where a function is defined within another). In the previous case:

```
>>> def func2():
...     global x
...     x += 1      # OK now - Python knows we mean x in global scope
...
>>> x = 4
>>> func2()         # No error
>>> x
5
```

The function `func2` really has changed the value of the variable `x` in global scope.

You should think carefully whether it is really necessary to use this technique (would it be better to pass x as an argument and `return` its updated value from the function?), Especially in longer programs, variable names in one scope that change value (or even type!) within functions lead to confusing code, behavior that is hard to predict and tricky bugs.

Example E2.28 Take a moment to study the following code and predict the result before running it.

Listing 2.5 Python scope rules

```
# eg2-scope.py

def outer_func():
    def inner_func():
        a = 9
        print('inside inner_func, a is {:d} (id={:d})'.format(a, id(a)))
        print('inside inner_func, b is {:d} (id={:d})'.format(b, id(b)))
        print('inside inner_func, len is {:d} (id={:d})'.format(len,id(len)))

    len = 2
    print('inside outer_func, a is {:d} (id={:d})'.format(a, id(a)))
    print('inside outer_func, b is {:d} (id={:d})'.format(b, id(b)))
    print('inside outer_func, len is {:d} (id={:d})'.format(len,id(len)))
    inner_func()

a, b = 6, 7
outer_func()
print('in global scope, a is {:d} (id={:d})'.format(a, id(a)))
print('in global scope, b is {:d} (id={:d})'.format(b, id(b)))
print('in global scope, len is', len, '(id={:d})'.format(id(len)))
```

This program defines a function, `inner_func` nested inside another, `outer_func`. After these definitions, the execution proceeds as follows:

1. Global variables `a=6` and `b=7` are initialized.
2. `outer_func` is called:
    a. `outer_func` defines a local variable, `len=2`.
    b. The values of `a` and `b` are printed; they don't exist in local scope and there isn't any enclosing scope, so Python searches for and finds them in global scope: their values (6 and 7) are output.
    c. The value of local variable `len` (2) is printed.
    d. `inner_func` is called:
        (1) A local variable, `a=9` is defined.
        (2) The value of this local variable is printed.
        (3) The value of `b` is printed; `b` doesn't exist in local scope so Python looks for it in enclosing scope, that of `outer_func`. It isn't found

there either, so Python proceeds to look in global scope where it is found: the value `b=7` is printed.

(4) The value of `len` is printed: `len` doesn't exist in local scope, but it is in the enclosing scope since `len=2` is defined in `outer_func`: its value is output.

3. After `outer_func` has finished execution, the values of `a` and `b` in global scope are printed.
4. The value of `len` is printed. This is not defined in global scope, so Python searches its own built-in names: `len` is the built-in function for determining the lengths of sequences. This function is itself an object and it provides a short string description of itself when printed.

```
inside outer_func, a is 6 (id=232)
inside outer_func, b is 7 (id=264)
inside outer_func, len is 2 (id=104)
inside inner_func, a is 9 (id=328)
inside inner_func, b is 7 (id=264)
inside inner_func, len is 2 (id=104)
in global scope, a is 6 (id=232)
in global scope, len is <built-in function len> (id=977)
```

Note that in this example `outer_func` has (perhaps unwisely) redefined (*re-bound*) the name `len` to the integer object `2`. This means that the original `len` built-in function is not available within this function (and neither is it available within the enclosed function, `inner_func`).

2.7.4 ◊ Passing arguments to functions

A common question from new users of Python who come to it with a knowledge of other computer languages is, are arguments to functions passed "by value" or "by reference?" In other words, does the function make its own copy of the argument, leaving the caller's copy unchanged, or does it receive a "pointer" to the location in memory of the argument, the contents of which the function *can* change? The distinction is important for languages such as C, but does not fit well into the Python *name-object* model. Python function arguments are sometimes (not very helpfully) said to be "references, passed by value." Recall that everything in Python is an object, and the same object may have multiple identifiers (what we have been loosely calling "variables" up until now). When a name is passed to a function, the "value" that is passed is, in fact, the it points to. Whether the function can change the object or not (from the point of view of the caller) depends on whether the object is mutable or immutable.

A couple of examples should make this clearer. A simple function, `func1`, taking an integer argument, receives a reference to that integer object, to which it attaches a local name (which may or may not be the same as the global name). The function cannot change the integer object (which is immutable), so any reassignment of the local name simply points to a new object: the global name still points to the original integer object.

```
>>> def func1(a):
...     print('func1: a = {}, id = {}'.format(a, id(a)))
...     a = 7 # reassigns local a to the integer 7
...     print('func1: a = {}, id = {}'.format(a, id(a)))
...
>>> a = 3
>>> print('global: a = {}, id = {}'.format(a, id(a)))

global: a = 3, id = 4297242592

>>> func1(a)
func1: a = 3, id = 4297242592
func1: a = 7, id = 4297242720

>>> print('global: a = {}, id = {}'.format(a, id(a)))

global: a = 3, id = 4297242592
```

`func1` therefore prints `3` (inside the function, `a` is initially the local name for the original integer object); it then prints `7` (this local name now points to a new integer object, with a new `id`) – see Figure 2.5. After it returns, the global name `a` still points to the original `3`.

Now consider passing a mutable object, such as a `list` to a function, `func2`. This time, an assignment to the list changes the original object, and these changes persist after the function call.

```
>>> def func2(b):
...     print('func2: b = {}, id = {}'.format(b, id(b)))
...     b.append(7) # add an item to the list
...     print('func2: b = {}, id = {}'.format(b, id(b)))
...
>>> c = [1, 2, 3]
>>> print('global: c = {}, id = {}'.format(c, id(c)))

global: c = [1, 2, 3], id = 4361122448

>>> func2(c)
func2: b = [1, 2, 3], id = 4361122448
func2: b = [1, 2, 3, 7], id = 4361122448
```

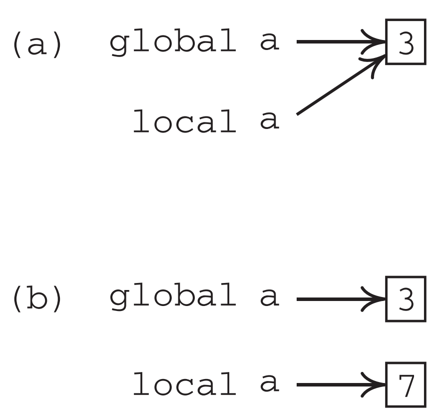


Figure 2.5 Immutable objects. Within `func1`: (a) before reassigning the local variable `a` and (b) after reassigning the value of local variable `a`.

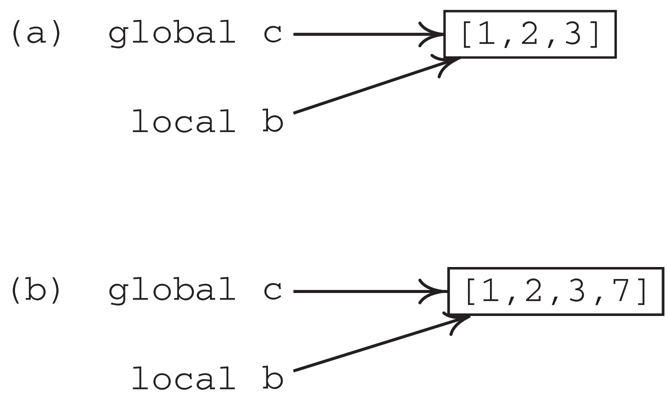


Figure 2.6 Mutable objects. Within `func2`: (a) before appending to the list pointed to by both global variable `c` and local variable `b` and (b) after appending to the list with `b.append(7)`.

```
>>> print('global: c = {}, id = {}'.format(c, id(c)))

global: c = [1, 2, 3, 7], id = 4361122448
```

Note that it doesn't matter what name is given to the list by the function: this name points to the same object, as you can see from its `id`. The relationships between the variable names and objects is illustrated in Figure 2.6.

So are Python arguments passed by value or by reference? The best answer is probably that arguments are passed by value, but that value is a reference to an object (which can be mutable or immutable).

Example E2.29 The *Lazy Caterer's Sequence*, $f(n)$, describes the maximum number of pieces a circular pizza can be divided into with an increasing number of cuts, n. Clearly $f(0) = 1$, $f(1) = 2$ and $f(2) = 4$). For $n = 3$, $f(3) = 7$ (the maximum number of pieces are formed if the cuts do not intesect at a common point). It can be shown that the general recursion formula,

$$f(n) = f(n-1) + n,$$

applies. Although there is a closed form for this sequence, $f(n) = \frac{1}{2}(n^2 + n + 2)$, we could also define a function to grow a list of consecutive values in the sequence:

```
>>> def f(seq):
...     seq.append(seq[-1] + n)
...
>>> seq = [1]     # f(0) = 1
>>> for n in range(1,16):
...     f(seq)
...
>>> print(seq)
[1, 2, 4, 7, 11, 16, 22, 29, 37, 46, 56, 67, 79, 92, 106, 121]
```

The list `seq` is mutable and so grows in place each time the function `f()` is called. The `n` referred to within this function is the name found in global scope (the `for` loop counter).

2.7.5 Recursive functions

A function that can call itself is called a *recursive* function. Recursion is not always necessary but can lead to elegant algorithms in some situations.[32] For example, one way to calculate the factorial of an integer $n \geq 1$ is to define the following recursive function:

```
>>> def factorial(n):
...     if n == 1:
...         return 1
...     return n * factorial(n-1)
...
>>> factorial(5)
120
```

Here, a call to `factorial(n)` returns n times whatever is returned by the call to `factorial(n-1)`, which returns $n-1$ times the returned values of `factorial(n-2)` and so on until `factorial(1)` which is 1 by definition. That is, the algorithm makes use of the fact that $n! = n \cdot (n-1)!$ Care should be taken in implementing such recursive algorithms to ensure that they stop when some condition is met.[33]

Example E2.30 The famous *Tower of Hanoi* problem involves three poles, one of which (pole A) is stacked with n differently sized, circular discs, in decreasing order of diameter with the largest at the bottom. The task is to move the stack to the third pole (pole C) by moving one disc at a time in such a way that a larger disc is never placed on a smaller one. It is necessary to use the second pole (pole B) as an intermediate resting place for the discs.

The problem can be solved using the following recursive algorithm. Label the discs D_i with D_1 the smallest disc and D_n the largest.

- Move discs $D_1, D_2, \cdots, D_{n-1}$ from A to B;
- Move disc D_n from A to C;
- Move discs $D_1, D_2, \cdots, D_{n-1}$ from B to C.

The second step is a single move, but the first and last require the movement of a stack of $n-1$ discs from one peg to another – which is exactly what the algorithm itself solves!

In the following code, we identify the discs by the integers $1, 2, 3, \cdots$ stored in one of three lists, `A`, `B` and `C`. The initial state of the system, with all discs on pole A is denoted by, for example, `A = [5,4,3,2,1]` where the first indexed item is the "bottom" of the pole and the last indexed item is the "top." The rules of the problem require that these lists must always be *decreasing* sequences.

---

[32] In fact, because of the overhead involved in making a function call, a recursive algorithm can be expected to be slower than a well-designed iterative one.

[33] In practice, an infinite loop is not possible because of the memory overhead involved in each function call, and Python sets a maximum recursion limit.

Listing 2.6 The Tower of Hanoi problem

```
# eg2-hanoi.py

def hanoi(n, P1, P2, P3):
    """ Move n discs from pole P1 to pole P3. """
    if n == 0:
        # No more discs to move in this step
        return

    global count
    count += 1

    # move n-1 discs from P1 to P2
    hanoi(n-1, P1, P3, P2)

    if P1:
        # move disc from P1 to P3
        P3.append(P1.pop())
        print(A, B, C)

    # move n-1 discs from P2 to P3
    hanoi(n-1, P2, P1, P3)

# Initialize the poles: all n discs are on pole A.
n = 3
A = list(range(n,0,-1))
B, C = [], []

print(A, B, C)
count = 0
hanoi(n, A, B, C)
print(count)
```

Note that the `hanoi` function just moves a stack of discs from one pole to another: lists (representing the poles) are passed into it in some order, and it moves the discs from the pole represented by the first list, known locally as `P1`, to that represented by the third (`P3`). It does not need to know which list is `A`, `B` or `C`.

2.7.6 Exercises

Questions

Q2.7.1 The following small programs each attempt to output the simple sum:

```
   56
  +44
-----
  100
-----
```

Which two programs work as intended? Explain carefully what is wrong with each of the others.

a.
```
def line():
    '-----'

my_sum = '\n'.join(['   56', '  +44', line(), '  100', line()])
print(my_sum)
```

b.
```
def line():
    return '-----'

my_sum = '\n'.join(['   56', '  +44', line(), '  100', line()])
print(my_sum)
```

c.
```
def line():
    return '-----'

my_sum = '\n'.join(['   56', '  +44', line, '  100', line])
print(my_sum)
```

d.
```
def line():
    print('-----')

print('   56')
print('  +44')
print(line)
print('  100')
print(line)
```

e.
```
def line():
    print('-----')

print('   56')
print('  +44')
print(line())
print('  100')
print(line())
```

f.
```
def line():
    print('-----')

print('   56')
print('  +44')
line()
print('  100')
line()
```

Q2.7.2 The following code snippet attempts to calculate the balance of a savings account with an annual interest rate of 5% after 4 years, if it starts with a balance of $100.

```
>>> balance = 100
>>> def add_interest(balance, rate):
...     balance += balance * rate / 100
...
>>> for year in range(4):
...     add_interest(balance, 5)
```

```
...     print('Balance after year {}: ${:.2f}'.format(year+1, balance))
...
Balance after year 1: $100.00
Balance after year 2: $100.00
Balance after year 3: $100.00
Balance after year 4: $100.00
```

Explain why this doesn't work and then provide a working alternative.

Q2.7.3 A *Harshad number* is an integer that is divisible by the sum of its digits (e.g., 21 is divisible by $2 + 1 = 3$ and so is a Harshad number). Correct the following code which should return `True` or `False` if `n` is a Harshad number or not respectively:

```
def digit_sum(n):
    """ Find the sum of the digits of integer n. """

    s_digits = list(str(n))
    dsum = 0
    for s_digit in s_digits:
        dsum += int(s_digit)

def is_harshad(n):
    return not n % digit_sum(n)
```

When run, the function `is_harshad` raises an error:

```
>>> is_harshad(21)
TypeError: unsupported operand type(s) for %: 'int' and 'NoneType'
```

Problems

P2.7.1 The word game Scrabble is played on a 15×15 grid of squares referred to by a row index letter (A – O) and a column index number (1 – 15). Write a function to determine whether a word will fit in the grid, given the position of its first letter as a string (e.g., `'G7'`) a variable indicating whether the word is placed to read *across* or *down* the grid and the word itself.

P2.7.2 Write a program to find the smallest positive integer, n, whose factorial is *not* divisible by the sum of its digits. For example, 6 is not such a number because $6! = 720$ and $7 + 2 + 0 = 9$ divides 720.

P2.7.3 Write two functions which, given two lists of length 3 representing three-dimensional vectors **a** and **b**, calculate the dot product, $\mathbf{a} \cdot \mathbf{b}$ and the vector (cross) product, $\mathbf{a} \times \mathbf{b}$.

Write two more functions to return the scalar triple product, $\mathbf{a} \cdot (\mathbf{b} \times \mathbf{c})$ and the vector triple product, $\mathbf{a} \times (\mathbf{b} \times \mathbf{c})$.

P2.7.4 A right regular pyramid with height h and a base consisting of a regular n-sided polygon of side length s has a volume, $V = \frac{1}{3}Ah$ and total surface area, $S = A + \frac{1}{2}nsl$ where A is the base area and l the slant height, which may be calculated from the *apothem* of the base polygon, $a = \frac{1}{2}s \cot\frac{\pi}{n}$ as $A = \frac{1}{2}nsa$ and $l = \sqrt{h^2 + a^2}$.

Use these formulas to define a function, `pyramid_AV`, returning V and S when passed values for n, s and h.

P2.7.5 The range of a projectile launched at an angle α and speed v on flat terrain is

$$R = \frac{v^2 \sin 2\alpha}{g},$$

where g is the acceleration due to gravity which may be taken to be $9.81\ \mathrm{m\,s^{-2}}$ for Earth. The maximum height attained by the projectile is given by

$$H = \frac{v^2 \sin^2 \alpha}{2g}.$$

(We neglect air resistance and the curvature and rotation of the Earth.) Write a function to calculate and return the range and maximum height of a projectile, taking α and v as arguments. Test it with the values $v = 10\ \mathrm{m\,s^{-1}}$ and $\alpha = 30°$.

P2.7.6 Write a function, `sinm_cosn`, which returns the value of the following definite integral for integers $m, n > 1$.

$$\int_0^{\pi/2} \sin^n \theta \cos^m \theta \, \mathrm{d}\theta = \begin{cases} \frac{(m-1)!!(n-1)!!}{(m+n)!!}\frac{\pi}{2} & m, n \text{ both even,} \\ \frac{(m-1)!!(n-1)!!}{(m+n)!!} & \text{otherwise.} \end{cases}$$

Hint: for calculating the double factorial, see Exercise P2.4.6.

P2.7.7 Write a function that determines if a string is a palindrome (that is, reads the same backward as forward) *using recursion*.

P2.7.8 *Tetration* may be thought of as the next operator after exponentiation: Thus, where $x \times n$ can be written as the sum $x + x + x + \cdots + x$ with n terms, and x^n is the multiplication of n factors: $x \cdot x \cdot x \cdots x$, the expression written ${}^n x$ is equal to the repeated exponentiation involving n occurrences of x:

$${}^n x = x^{x^{\cdot^{\cdot^{x}}}}$$

For example, ${}^4 2 = 2^{2^{2^2}} = 2^{2^4} = 2^{16} = 65536$. Note that the exponential "tower" is evaluated from top to bottom.

Write a recursive function to calculate ${}^n x$ and test it (for small, positive real values of x and non-negative integers n: tetration generates *very* large numbers)!

How many digits are there in ${}^3 5$? In ${}^5 2$?

3 Interlude: simple plotting with `pylab`

As Python has grown in popularity, many libraries of packages and modules have become available to extend its functionality in useful ways; Matplotlib is one such library. Matplotlib provides a means of producing graphical plots that can be embedded into applications, displayed on the screen or output as high-quality image files for publication.

Matplotlib has a fully fledged *object-oriented* interface, which is described in more detail in Chapter 7, but for simple plotting in an interactive shell session, its simpler, *procedural* `pylab` interface provides a convenient way of visualizing data. `pylab` is designed to be easy to learn and functions in a similar way to comparable tools in the commercial MATLAB package.

On a system with Matplotlib installed the `pylab` package is imported with

```
>>> import pylab
```

even though this means prefacing all of the `pylab` method calls with "`pylab.`"[1]

3.1 Basic plotting

3.1.1 Line plots and scatterplots

The simplest (x, y) line plot is achieved by calling `pylab.plot` with two iterable objects of the same length (typically lists of numbers or NumPy arrays). For example,

```
>>> ax = [0., 0.5, 1.0, 1.5, 2.0, 2.5, 3.0]
>>> ay = [0.0, 0.25, 1.0, 2.25, 4.0, 6.25, 9.0]
>>> pylab.plot(ax,ay)
>>> pylab.show()
```

`pylab.plot` creates a matplotlib object (here, a `Line2D` object) and `pylab.show()` displays it on the screen. Figure 3.1 shows the result; by default the line will be in blue.

To plot (x, y) points as a scatterplot rather than as a line plot, call `pylab.scatter` instead:

```
>>> import random
>>> ax, ay = [], []
```

[1] It is better to avoid polluting the global namespace by importing as `from pylab import *`.

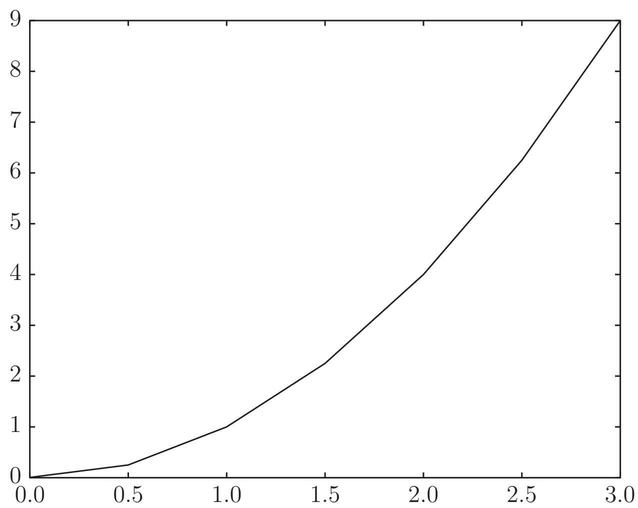


Figure 3.1 A basic (x, y) line plot.

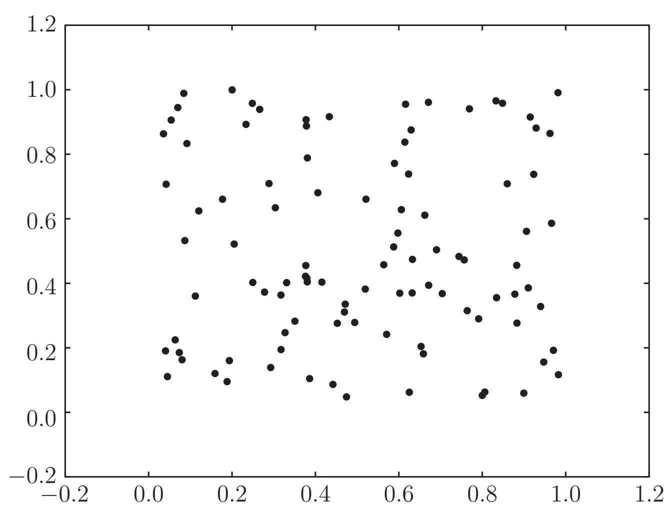


Figure 3.2 A basic scatter plot.

```
>>> for i in range(100):
...     ax.append(random.random())
...     ay.append(random.random())
...
>>> pylab.scatter(ax,ay)
>>> pylab.show()
```

The resulting plot is shown in Figure 3.2.

We can also save the plot as an image by calling `pylab.savefig(filename)`. The desired image format is deduced from the filename extension. For example,

```
pylab.savefig('plot.png')        # save as a PNG image
pylab.savefig('plot.pdf')        # save as PDF
pylab.savefig('plot.eps')        # save in Encapsulated PostScript format
```

Example E3.1 As an example, let's plot the function $y = \sin^2 x$ for $-2\pi \leq x \leq 2\pi$. Using only the Python we've covered in the previous chapter, here is one approach:

We calculate and plot 1,000 (x, y) points, and store them in the lists `ax` and `ay`. To set up the `ax` list as the abcissa, we can't use `range` directly because that method only produces integer sequences, so first we work out the spacing between each x value as

$$\Delta x = \frac{x_{\text{max}} - x_{\text{min}}}{n - 1}$$

(if our n values are to *include* x_{min} and x_{max}, there are $n - 1$ intervals of width Δx); the abcissa points are then

$$x_i = x_{\text{min}} + i\Delta x \quad \text{for } i = 0, 1, 2, \cdots, n - 1.$$

The corresponding y-axis points are

$$y_i = \sin^2(x_i).$$

The following program implements this approach, and plots the (x, y) points on a simple line-graph (see Figure 3.3).

Listing 3.1 Plotting $y = \sin^2 x$

```
# eg3-sin2x.py

import math
import pylab
xmin, xmax = -2. * math.pi, 2. * math.pi
n = 1000
x = [0.] * n
y = [0.] * n
dx = (xmax - xmin)/(n-1)
for i in range(n):
    xpt = xmin + i * dx
    x[i] = xpt
    y[i] = math.sin(xpt)**2

pylab.plot(x,y)
pylab.show()
```

3.1.2 `linspace` and vectorization

Plotting the simple function $y = \sin^2 x$ in the previous example involved quite a lot of work, almost all of it to do with setting up the lists `x` and `y`. In fact, `pylab` provides some of the same functionality as the NumPy library introduced in Chapter 6, which can be used to make life much easier.

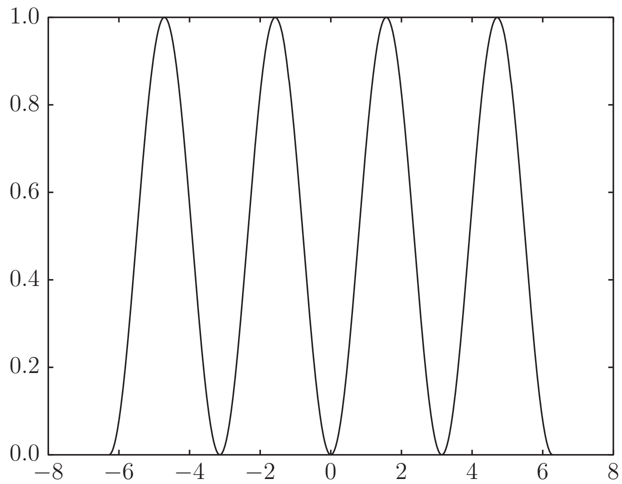


Figure 3.3 A plot of $y = \sin^2 x$.

First, the regularly spaced grid of x-coordinates, `x`, can be created using `linspace`. This is much like a floating point version of the `range` built-in: it takes a start value, an end value, and the number of values in the sequence and generates an array of values representing the arithmetic progression between (and *inclusive of*) the two values. For example, `x = pylab.linspace(-5, 5, 1001)` creates the sequence: $-5.0, -4.99, -4.98, ..., 4.99, 5.0$.

Second, the `pylab` equivalents of the `math` module's methods can act on iterable objects (such as `lists` or NumPy arrays). Thus, `y = pylab.sin(x)` creates a sequence of values (actually, a NumPy `ndarray`), which are $\sin(x_i)$ for each value x_i in the array `x`:

```
import pylab
n = 1000
xmin, xmax = -2. * math.pi, 2. * math.pi
x = pylab.linspace(xmin, xmax, n)
y = pylab.sin(x)**2
pylab.plot(x,y)
pylab.show()
```

This is called *vectorization* and is described in more detail in Section 6.1.3. Lists and tuples can be turned into array objects supporting vectorization with the `array` constructor method:

```
>>> w = [1.0, 2.0, 3.0, 4.0]
>>> w = pylab.array(w)
>>> w * 100     # multiply each element by 100
array([ 100.,  200.,  300.,  400.])
```

To add a second line to the plot, simply call `pylab.plot` again:

```
...
x = pylab.linspace(xmin, xmax, n)
y1 = pylab.sin(x)**2
```

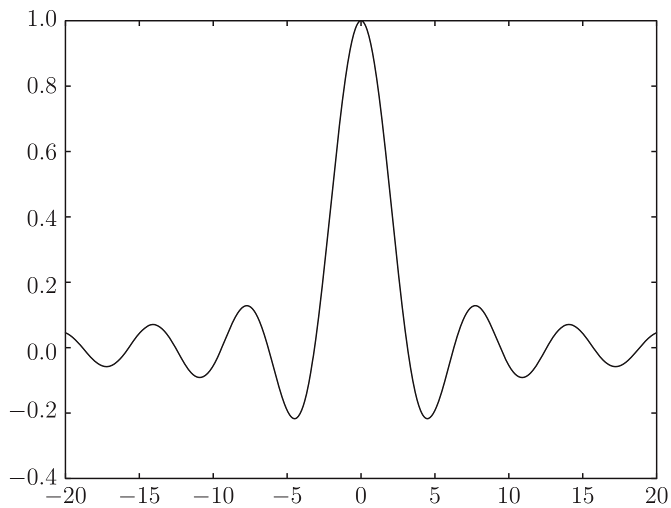


Figure 3.4 A plot of $y = \text{sinc}(x)$.

```
y2 = pylab.cos(x)**2
pylab.plot(x,y1)
pylab.plot(x,y2)
pylab.show()
```

Note that after a plot has been displayed with `show` or saved with `savefig`, it is no longer available to display a second time – to do this it is necessary to call `pylab.plot` again. This is because of the procedural nature of the `pylab` interface: each call to a `pylab` method changes the internal *state* of the plot object. The plot object is built up by successive calls to such methods (adding lines, legends and labels, setting the axis limits, etc.), and then the plot object is displayed or saved.

Example E3.2 The sinc function is the function

$$f(x) = \frac{\sin x}{x}.$$

To plot it over $20 \le x \le 20$:

```
>>> x = pylab.linspace(-20, 20, 1001)
>>> y = pylab.sin(x)/x

__main__:1: RuntimeWarning: invalid value encountered in true_divide
>>> pylab.plot(x,y)
>>> pylab.show()
```

Note that even though Python warns of the division by zero at $x = 0$, the function is plotted correctly: the singular point is set to the special value `nan` (standing for "not a number") and is omitted from the plot (Figure 3.4).

```
>>> y[498:503]
array([ 0.99893367,  0.99973335,         nan,  0.99973335,  0.99893367])
```

3.1.3 Exercises

Problems

P3.1.1 Plot the functions

$$f_1(x) = \ln\left(\frac{1}{\cos^2 x}\right) \text{ and}$$
$$f_2(x) = \ln\left(\frac{1}{\sin^2 x}\right).$$

on 1,000 points across the range $-20 \le x \le 20$. What happens to these functions at $x = n\pi/2$ $(n = 0, \pm 1, \pm 2, \ldots)$? What happens in your plot of them?

P3.1.2 The *Michaelis-Menten* equation models the kinetics of enzymatic reactions as

$$v = \frac{\mathrm{d[P]}}{\mathrm{d}t} = \frac{V_{\max}[\mathrm{S}]}{K_m + [\mathrm{S}]},$$

where v is the rate of the reaction converting the substrate, S, to product P, catalyzed by the enzyme. $V_{\max}$ is the maximum rate (when all the enzyme is bound to S) and the Michaelis constant, K_m, is the substrate concentration at which the reaction rate is at half its maximum value.

Plot v against [S] for a reaction with $K_m = 0.04$ M and $V_{\max} = 0.1\ \mathrm{M\,s^{-1}}$. Look ahead to the next section if you want to label the axes.

P3.1.3 The normalized Gaussian function centered at $x = 0$ is

$$g(x) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left(\frac{x^2}{2\sigma^2}\right).$$

Plot and compare the shapes of these functions for standard deviations $\sigma = 1, 1.5$ and 2.

3.2 Labels, legends and customization

3.2.1 Labels and legends

Plot legend

Each line on a `pylab` plot can be given a label by passing a string object to its `label` argument. However, the label won't appear on the plot unless you also call `pylab.legend` to add a legend:

```
pylab.plot(ax, ay1, label='sin^2(x)')
pylab.legend()
pylab.show()
```

The location of the legend is, by default, the top right-hand corner of the plot but can be customized by setting the `loc` argument to the `legend` method to either the string or integer values given in Table 3.1.

Table 3.1 Legend location specifiers

String	Integer
'best'	0
'upper right'	1
'upper left'	2
'lower left'	3
'lower right'	4
'right'	5
'center left'	6
'center right'	7
'lower center'	8
'upper center'	9
'center'	10

The plot title axis labels

A plot can be given a title above the axes by calling `pylab.title` and passing the title as a string. Similarly, the methods `pylab.xlabel` and `pylab.ylabel` control the labeling of the x- and y-axes: just pass the label you want as a string to these methods. The optional additional attribute `fontsize` sets the font size in points. For example,

```
t = pylab.linspace(0., 0.1, 1000)
Vp_uk, Vp_us = 230, 110
f_uk, f_us = 50, 60
❶ V_uk = Vp_uk * pylab.sin(2 * pylab.pi * f_uk * t)
V_us = Vp_us * pylab.sin(2 * pylab.pi * f_us * t)
❷ pylab.plot(t*1000, V_uk, label='UK')
pylab.plot(t*1000, V_us, label='US')
pylab.title('A comparison of AC voltages in the UK and US')
pylab.xlabel('Time /ms', fontsize=16.)
pylab.ylabel('Voltage /V', fontsize=16.)
pylab.legend()
pylab.show()
```

❶ We calculate the voltage as a function of time (`t`, in seconds) in the United Kingdom and in the United States, which have different peak voltages (230 V and 110 V respectively) and different frequencies (50 Hz and 60 Hz).
❷ The time is plotted on the x-axis in milliseconds (`t*1000`) – see Figure 3.5.

Using LaTeX in `pylab`

You can use LaTeX markup in `pylab` plots, but this option needs to be enabled in Matplotlib's "rc settings," as follows:

```
pylab.rc('text', usetex=True)
```

Then simply pass the LaTeX markup as a string to any label you want displayed in this way. Remember to use raw strings (`r'xxx'`) to prevent Python from escaping any characters followed by LaTeX's backslashes (see Section 2.3.2).

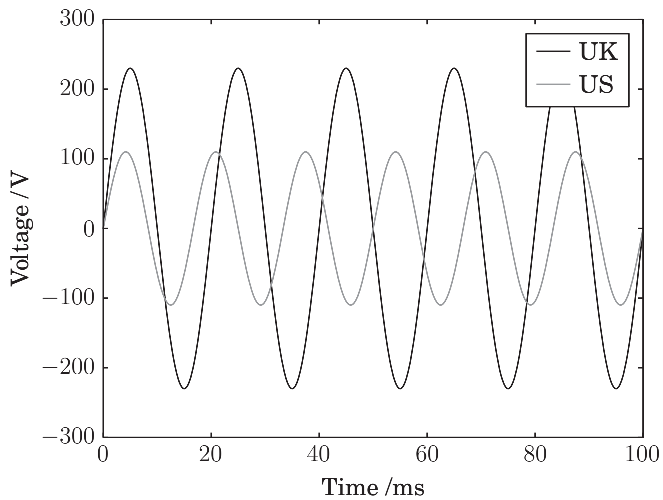


Figure 3.5 A comparison of AC voltages in the United Kingdom and United States.

Example E3.3 To plot the functions $f_n(x) = x^n \sin x$ for $n = 1, 2, 3, 4$:

```
import pylab
pylab.rc('text', usetex=True)

x = pylab.linspace(-10,10,1001)
for n in range(1,5):
    y = x**n * pylab.sin(x)
❶   y /= max(y)
    pylab.plot(x,y, label=r'$x^{}\sin x$'.format(n))
pylab.legend(loc='lower center')
pylab.show()
```

❶ To make the graphs easier to compare, they have been scaled to a maximum of 1 in the region considered.

The graph produced is given in Figure 3.6.

3.2.2 Customizing plots

Markers

By default, `plot` produces a line-graph with no markers at the plotted points. To add a marker on each point of the plotted data, use the `marker` argument. Several different markers are available and are documented online;[2] some of the more useful ones are listed in Table 3.2.

Colors

The color of a plotted line and/or its markers can be set with the `color` argument. Several formats for specifying the color are supported. First, there are one-letter codes

[2] http://matplotlib.org/api/markers_api.htmlmodule-matplotlib.markers.

Table 3.2 Some Matplotlib marker styles

Code	Marker	Description
`.`	·	point
`o`	○	circle
`+`	+	plus
`x`	×	x
`D`	◇	diamond
`v`	▽	downward triangle
`^`	△	upward triangle
`s`	□	square
`*`	⋆	star

Table 3.3 Matplotlib color code letters

Code	Color
`b`	blue
`g`	green
`r`	red
`c`	cyan
`m`	magenta
`y`	yellow
`k`	black
`w`	white

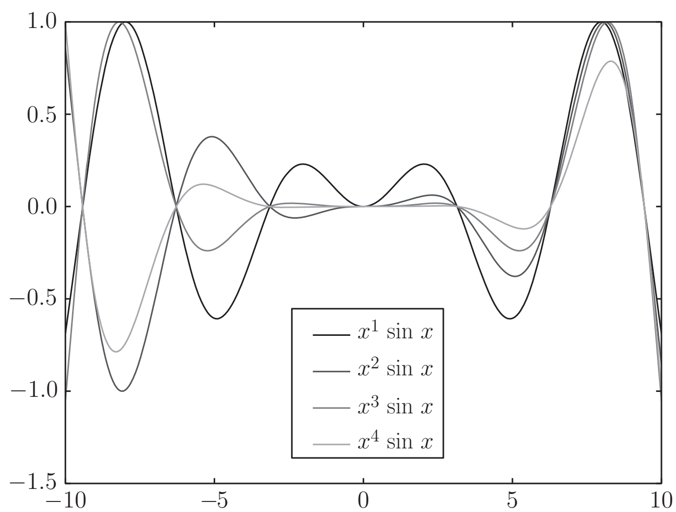


Figure 3.6 $f_n(x) = x^2 \sin x$ for $n = 1, 2, 3, 4$.

for some common colors, given in Table 3.3. For example, `color='r'` specifies a red line and markers. The default color sequence for a series of lines on the same plot is in the same order as this table.

Alternatively, shades of gray can specified as a string representing a `float` in the range 0–1 (`'0.'` being black and `'1.'` being white). HTML hex strings giving the

Table 3.4 Matplotlib line styles

Code	Line style
`-`	solid
`--`	dashed
`:`	dotted
`-.`	dash-dot

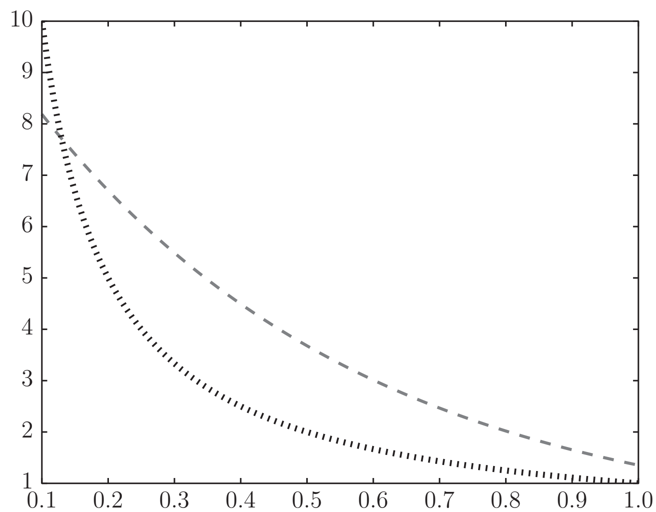


Figure 3.7 Two different line styles on the same plot.

red, green and blue (RGB) components of the color in the range `00` – `ff` can also be passed in the `color` argument (e.g., `color='#ff00ff'` is magenta). Finally, the RGB components can also be passed as a `tuple` of three values in the range 0–1 (e.g., `color=(0.5, 0., 0.)` is a dark red color).

Line styles and widths

The default plot line style is a solid line of weight 1.0 pt. To customize this, set the `linestyle` argument (also a string). Some of the possible line style settings are given in Table 3.4.

To draw no line at all, set `linestyle=''` (the empty string). The thickness of a line can be specified in points by passing a `float` to the `linewidth` attribute.

For example,

```
ax = pylab.linspace(0.1, 1., 100)
ayi = 1./ax
aye = 10. * pylab.exp(-2.*ax)
pylab.plot(ax, ayi, color='r', linestyle=':', linewidth=4.)
pylab.plot(ax, aye, color='m', linestyle='--', linewidth=2.)
pylab.show()
```

This code produces Figure 3.7.

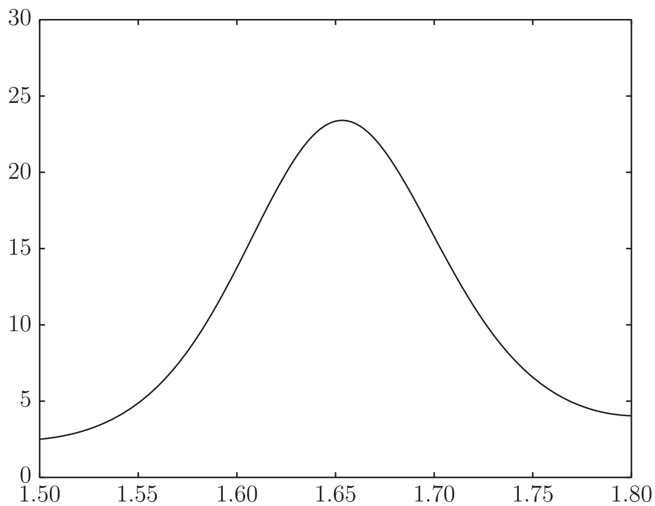


Figure 3.8 A plot produced with explicitly defined data limits.

The following abbreviations for the plot line properties are also valid:

- `c` for `color`
- `ls` for `linestyle`
- `lw` for `linewidth`

For example,

```
pylab.plot(x, y, c='g', ls='--', lw=2)      # a thick, green, dashed line
```

It is also possible to specify the color, linestyle and marker style in a single string:

```
pylab.plot(x, y, 'r:^')      # a red, dotted line with triangle markers
```

Finally, multiple lines can be plotted using a sequence of `x`, `y`, `format` arguments:

```
pylab.plot(x,y1, 'r--', x, y2, 'k-.')
```

plots a red dashed line for `(x,y1)` and a black dash-dot line for `(x,y2)`.

Plot limits

The methods `pylab.xlim` and `pylab.ylim` set the *x*- and *y*- limits of the plot respectively. They must be called *after* any `pylab.plot` statements, before showing or saving the figure. For example, the following code produces a plot of the provided data series between chosen limits (Figure 3.8):

```
t = pylab.linspace(0, 2, 1000)
f = t * pylab.exp(t + pylab.sin(20*t))
pylab.plot(t, f)
pylab.xlim(1.5,1.8)
pylab.ylim(0,30)
pylab.show()
```

Example E3.4 *Moore's Law* is the observation that the number of transistors on CPUs approximately doubles every two years. The following program illustrates this with

a comparison between the actual number of transistors on high-end CPUs from between 1972 and 2012, and that predicted by Moore's Law which may be stated mathematically as:

$$n_i = n_0 2^{(y_i - y_0)/T_2},$$

where n_0 is the number of transistors in some reference year, y_0, and $T_2 = 2$ is the number of years taken to double this number. Because the data cover 40 years, the values of n_i span many orders of magnitude, and it is convenient to apply Moore's Law to its logarithm, which shows a linear dependence on y:

$$\log_{10} n_i = \log_{10} n_0 + \frac{y_i - y_0}{T_2} \log_{10} 2.$$

Listing 3.2 An illustration of Moore's Law

```
# eg3-moore.py
import pylab

# The data - lists of years:
year = [1972, 1974, 1978, 1982, 1985, 1989, 1993, 1997, 1999, 2000, 2003,
        2004, 2007, 2008, 2012]
# and number of transistors (ntrans) on CPUs in millions:
ntrans = [0.0025, 0.005, 0.029, 0.12, 0.275, 1.18, 3.1, 7.5, 24.0, 42.0,
          220.0, 592.0, 1720.0, 2046.0, 3100.0]
# turn the ntrans list into a pylab array and multiply by 1 million
ntrans = pylab.array(ntrans) * 1.e6

y0, n0 = year[0], ntrans[0]
# A linear array of years spanning the data's years
y = pylab.linspace(y0, year[-1], year[-1] - y0 + 1)
# Time taken in years for the number of transistors to double
T2 = 2.
moore = pylab.log10(n0) + (y - y0) / T2 * pylab.log10(2)

pylab.plot(year, pylab.log10(ntrans), '*', markersize=12, color='r',
           markeredgecolor='r', label='observed')
pylab.plot(y, moore, linewidth=2, color='k', linestyle='--',
           label='predicted')
pylab.legend(fontsize=16, loc='upper left')
pylab.xlabel('Year', fontsize=16)
pylab.ylabel('log(ntrans)', fontsize=16)
pylab.title("Moore's Law")
pylab.show()
```

In this example, the data are given in two lists of equal length representing the year and representative number of transistors on a CPU in that year. The Moore's Law formula above is implemented in logarithmic form, using an array of years spanning the provided data. (Actually, since on a logarithmic scale this will be a straight line, really only two points are needed.)

For the plot, shown in Figure 3.9, the data are plotted as largeish stars and the Moore's Law prediction as a thick black line.

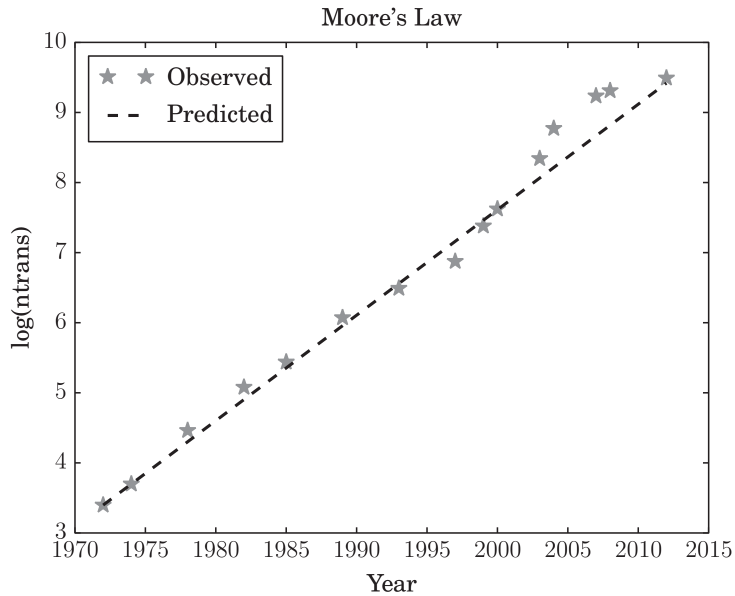


Figure 3.9 Moore's Law.

3.2.3 Exercises

Problems

P3.2.1 A molecule, A, reacts to form either B or C with first-order rate constants k_1 and k_2 respectively. That is,

$$\frac{\mathrm{d}[\mathrm{A}]}{\mathrm{d}t} = -(k_1 + k_2)[\mathrm{A}],$$

and so

$$[\mathrm{A}] = [\mathrm{A}]_0 \mathrm{e}^{-(k_1+k_2)t},$$

where $[\mathrm{A}]_0$ is the initial concentration of A. The product concentrations (starting from 0), increase in the ratio $[\mathrm{B}]/[\mathrm{C}] = k_1/k_2$ and conservation of matter requires $[\mathrm{B}] + [\mathrm{C}] = [\mathrm{A}]_0 - [\mathrm{A}]$. Therefore,

$$[\mathrm{B}] = \frac{k_1}{k_1 + k_2}[\mathrm{A}]_0 \left(1 - \mathrm{e}^{-(k_1+k_2)t}\right)$$

$$[\mathrm{C}] = \frac{k_2}{k_1 + k_2}[\mathrm{A}]_0 \left(1 - \mathrm{e}^{-(k_1+k_2)t}\right)$$

For a reaction with $k_1 = 300\ \mathrm{s}^{-1}$ and $k_2 = 100\ \mathrm{s}^{-1}$, plot the concentrations of A, B and C against time given an initial concentration of reactant, $[\mathrm{A}]_0 = 2.0\ \mathrm{mol\,dm}^{-3}$.

P3.2.2 A *Gaussian integer* is a complex number whose real and imaginary parts are both integers. A *Gaussian prime* is a Gaussian integer $x + iy$ such that either:

- one of x and y is zero and the other is a prime number of the form $4n + 3$ or $-(4n + 3)$ for some integer $n \geq 0$; or
- both x and y are nonzero and $x^2 + y^2$ is prime.

Consider the sequence of Gaussian integers traced out by an imaginary particle, initially at c_0, moving in the complex plane according to the following rule: it takes integer steps in its current direction (± 1 in either the real or imaginary direction), but turns *left* if it encounters a Gaussian prime. Its initial direction is in the positive real direction ($\Delta c = 1 + 0i \Rightarrow \Delta x = 1, \Delta y = 0$). The path traced out by the particle is called a *Gaussian prime spiral*.

Write a program to plot the Gaussian prime spiral starting at $c_0 = 5 + 23i$.

P3.2.3 The annual risk of death (given as "1 in N") for men and women in the UK in 2005 for different age ranges is given in the table below. Use `pylab` to plot these data on a single chart.

Age range	Female	Male
< 1	227	177
1–4	5376	4386
5–14	10417	8333
15–24	4132	1908
25–34	2488	1215
35–44	1106	663
45–54	421	279
55–64	178	112
65–74	65	42
75–84	21	15
> 84	7	6

3.3 More advanced plotting

3.3.1 Polar plots

`pylab.plot` produces a plot on Cartesian (x, y) axes. To produce a polar plot using (r, θ) coordinates, use `pylab.polar`, which is passed arguments `theta` (which is usually the independent variable) and `r`.

Example E3.5 A cardioid is the plane figure described in polar coordinates by $r = 2a(1 + \cos\theta)$ for $0 \leq \theta \leq 2\pi$:

```
theta = pylab.linspace(0, 2.*pylab.pi, 1000)
a = 1.
r = 2 * a * (1. + pylab.cos(theta))
pylab.polar(theta, r)
pylab.show()
```

The polar graph plotted by this code is illustrated in Figure 3.10.

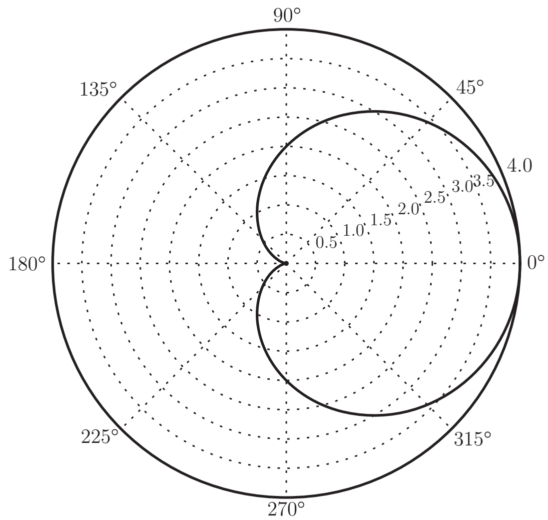


Figure 3.10 The cardioid figure formed with $a = 1$.

3.3.2 Histograms

A *histogram* represents the distribution of data as a series of (usually vertical) bars with lengths in proportion to the number of data items falling into predefined ranges (known as *bins*). That is, the range of data values is divided into intervals and the histogram constructed by counting the number of data values in each interval.

The `pylab` function `hist` produces a histogram from a sequence of data values. The number of bins can be passed as an optional argument, `bins`; its default value is 10. Also by default the height of the histogram bars are absolute counts of the data in the corresponding bin; setting the attribute `normed=True` normalizes the histogram so that its area (the height times width of each bar summed over the total number of bars) is unity.

For example, take 5,000 random values from the normal distribution with mean 0 and standard deviation 2 (see Section 4.5.1):

```
>>> import pylab
>>> import random
>>> data = []
>>> for i in range(5000):
...     data.append(random.normalvariate(0, 2))
>>> pylab.hist(data, bins=20, normed=True)
>>> pylab.show()
```

The resulting historgram is plotted in Figure 3.11.

3.3.3 Multiple axes

The command `pylab.twinx()` starts a new set of axes with the same x-axis as the original one, but a new y-scale. This is useful for plotting two or more data series, which share an abcissa (x-axis) but with y values which differ widely in magnitude or which have different units. This is illustrated in the following example.

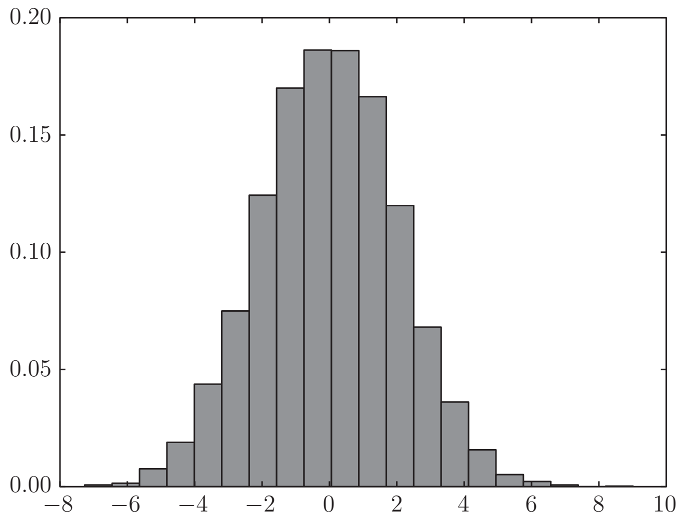


Figure 3.11 A histogram of random, normally distributed data.

Example E3.6 As described at http://tylervigen.com/, there is a curious but utterly meaningless correlation over time between the divorce rate in the US state of Maine and the per capita consumption of margarine in that country. The two time series here have different units and meanings and so should be plotted on separate y-axes, sharing a common x-axis (year).

Listing 3.3 The correlation between margarine consumption in the United States and the divorce rate in Maine

```
# eg3-margarine-divorce.py
import pylab

years = range(2000, 2010)
divorce_rate = [5.0, 4.7, 4.6, 4.4, 4.3, 4.1, 4.2, 4.2, 4.2, 4.1]
margarine_consumption = [8.2, 7, 6.5, 5.3, 5.2, 4, 4.6, 4.5, 4.2, 3.7]

❶ line1 = pylab.plot(years, divorce_rate, 'b-o',
                   label='Divorce rate in Maine')
pylab.ylabel('Divorces per 1000 people')
pylab.legend()

pylab.twinx()
line2 = pylab.plot(years, margarine_consumption, 'r-o',
                   label='Margarine consumption')
pylab.ylabel('lb of Margarine (per capita)')

# Jump through some hoops to get the both line's labels in the same legend:
❷ lines = line1 + line2
labels = []
for line in lines:
❸     labels.append(line.get_label())

pylab.legend(lines, labels)
pylab.show()
```

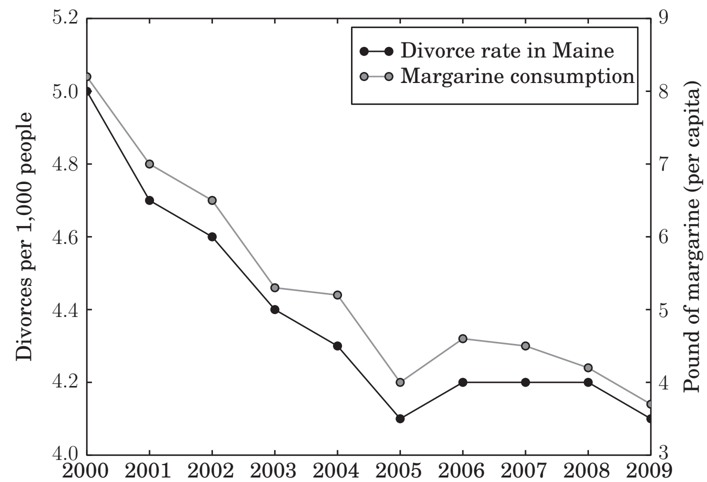


Figure 3.12 The correlation between the divorce rate in Maine and the per capita margarine consumption in the United States.

We have a bit of extra work to do in order to place a legend labeled with both lines on the plot: ❶ `pylab.plot` returns a list of objects representing the lines that are plotted, so we save them as `line1` and `line2`, ❷ concatenate them, and then ❸ loop over them to retrieve their labels. The list of lines and labels can then be passed to `pylab.legend` directly. The result of this code is the graph plotted in Figure 3.12.

3.3.4 Exercises

Problems

P3.3.1 A spiral may be considered to be the figure described by the motion of a point on an imaginary line as that line pivots around an origin at constant angular velocity. If the point is fixed on the line, then the figure described is a circle.

a. If the point on the rotating line moves from the origin with constant speed, its position describes an *Archimedean spiral*. In polar coordinates the equation of this spiral is $r = a + b\theta$. Use `pylab` to plot the spiral defined by $a = 0, b = 2$ for $0 \leq \theta \leq 8\pi$.

b. If the point moves along the rotating line with a velocity that increases in proportion to its distance from the origin, the result is a *logarithmic spiral*, which may be written as $r = a^{\theta}$. Plot the logarithmic spiral defined by $a = 0.8$ for $0 \leq \theta \leq 8\pi$. The logarithmic spiral has the property of *self-similarity*: with each 2π whorl, the spiral grows but maintains its shape.[3] Logarithmic spirals occur

[3] The Swiss mathematician Jakob Bernoulli was so taken with this property that he coined the logarithmic spiral *Spira mirabilis*: the "miraculus sprial" and wanted one engraved on his headstone with the phrase "Eadem mutata resurgo" ("Although changed, I shall arise the same"). Unfortunately, an Archimedian spiral was engraved by mistake.

frequently in nature, from the arrangements of the chambers of nautilus shells to the shapes of galaxies.

P3.3.2 A simple model for the interaction potential between two atoms as a function of their distance, r, is that of Lennard-Jones:

$$U(r) = \frac{B}{r^{12}} - \frac{A}{r^6},$$

where A and B are positive constants.[4]

For Argon atoms, these constants may be taken to be $A = 1.024 \times 10^{-23}\ \mathrm{J\,nm^6}$ and $B = 1.582 \times 10^{-26}\ \mathrm{J\,nm^{12}}$.

a. Plot $U(r)$. On a second y-axis on the same figure, plot the interatomic force

$$F(r) = -\frac{\mathrm{d}U}{\mathrm{d}r} = \frac{12B}{r^{13}} - \frac{6A}{r^7}.$$

Your plot should show the "interesting" part of these curves, which tend rapidly to very large values at small r.
Hint: life is easier if you divide A and B by Boltzmann's constant, $1.381 \times 10^{-23}\ \mathrm{J\,K^{-1}}$ so as to measure $U(r)$ in units of K. What is the depth, ϵ, and location, r_0, of the potential minimum for this system?

b. For small displacements from the equilibrium interatomic separation (where $F = 0$), the potential may be approximated to the harmonic oscillator function, $V(r) = \frac{1}{2}k(r - r_0)^2 + \epsilon$, where

$$k = \left|\frac{\mathrm{d}^2 U}{\mathrm{d}r^2}\right|_{r_0} = \frac{156B}{r_0^{14}} - \frac{42A}{r_0^8}.$$

Plot $U(r)$ and $V(r)$ on the same diagram.

P3.3.3 The seedhead of a sunflower may be modeled as follows. Number the n seeds $s = 1, 2, \cdots, n$ and place each seed a distance $r = \sqrt{s}$ from the origin, rotated $\theta = 2\pi s/\phi$ from the x axis, where ϕ is some constant. The choice nature makes for ϕ is the *golden ratio*, $\phi = (1 + \sqrt{5})/2$, which maximizes the packing efficiency of the seeds as the seedhead grows.

Write a Python program to plot a model sunflower seedhead. (*Hint:* use polar coordinates.)

[4] This was popular in the early days of computing because r^{-12} is easy to compute as the square of r^{-6}.

4 The core Python language II

This chapter continues the introduction to the core Python language started in Chapter 2 with a description of Python error handling with exceptions, the data structures known as dictionaries and sets, some convenient and efficient idioms to achieve common tasks, and a survey of some of the modules provided in the Python standard library. Finally, we present a brief introduction to *object-oriented programming* with Python.

4.1 Errors and exceptions

Python distinguishes between two types of error: *Syntax errors* and other *exceptions*. Syntax errors are mistakes in the grammar of the language and are checked for before the program is executed. Exceptions are *runtime* errors: conditions usually caused by attempting an invalid operation on an item of data. The distinction is that syntax errors are always fatal: there is nothing the Python compiler can do for you if your program does not conform to the grammar of the language. Exceptions, however, are conditions that arise during the running of a Python program (such as division by zero) and a mechanism exists for "catching" them and handling the condition gracefully without stopping the program's execution.

4.1.1 Syntax errors

Syntax errors are caught by the Python compiler and produce a message indicating where the error occurred. For example,

```
>>> for lambda in range(8):
  File "<stdin>", line 1
    for lambda in range(8):
             ^
SyntaxError: invalid syntax
```

Because `lambda` is a reserved keyword, it cannot be used as a variable name. Its occurrence where a variable name is expected is therefore a syntax error. Similarly,

```
>>> for f in range(8:
  File "<stdin>", line 1
    for f in range(8:
                    ^
SyntaxError: invalid syntax
```

The syntax error here occurs because a single argument to the `range` built-in must be given as an integer between parentheses: the colon breaks the syntax of calling functions and so Python complains of a syntax error.

Because a line of Python code may be split within an open bracket ("`()`", "`[]`", or "`{}`"), a statement split over several lines can sometimes cause a `SyntaxError` to be indicated somewhere other than the location of the true bug. For example,

```
>>> a = [1, 2, 3, 4,
... b = 5
  File "<stdin>", line 4
    b = 5
    ^
SyntaxError: invalid syntax
```

Here, the statement `b = 5` is syntactically valid: the error arises from failing to close the square bracket of the previous list declaration (the Python shell indicates that a line is a continuation of a previous one with the initial ellipsis ("`...`").

There are two special types of `SyntaxError` that are worth mentioning: an `IndentationError` occurs when a block of code is improperly indented and `TabError` is raised when a tabs and spaces are mixed inconsistently to provide indentation.[1]

Example E4.1 A common syntax error experienced by beginner Python programmers is in using the assignment operator "`=`" instead of the equality operator "`==`" in a conditional expression:

```
>>> if a = 5:
  File "<stdin>", line 1
    if a = 5:
         ^
SyntaxError: invalid syntax
```

This assignment `a = 5` does not return a value (it simply assigns the integer object `5` to the variable name `a`) and so there is nothing corresponding to `True` or `False` that the `if` statement can use: hence the `SyntaxError`. This contrasts with the C language in which an assignment returns the value of the variable being assigned (and so the statement `a = 5` evaluates to `true`). This behavior is the source of many hard-to-find bugs and security vulnerabilities and its omission from the Python language is by design.

4.1.2 Exceptions

An exception occurs when an syntactically correct expression is executed and causes a *runtime error*. There are different types of built-in exception, and custom exceptions can be defined by the programmer if required. If an exception is not "caught" using the `try ... except` clause described later, Python produces a (usually helpful) error message. If the exception occurs within a function (which may have been called, in turn, by

[1] This error can be avoided by using only spaces to indent code.

another function, and so on), the message returned takes the form of a *stack traceback*: the history of function calls leading to the error is reported so that its location in the program execution can be determined.

Some built-in exceptions will be familiar from your use of Python so far.

NameError

```
>>> print('4z = ', 4*z)

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'z' is not defined
```

A `NameError` exception occurs when a variable name is used that hasn't been defined: the `print` statement here is valid, but Python doesn't know what to print for `z`.

ZeroDivisionError

```
>>> a, b = 0, 5
>>> b / a

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: float division by zero
```

Division by zero is not mathematically defined.

TypeError and ValueError

A `TypeError` is raised if an object of the wrong type is used in an expression or function. For example,

```
>>> '00' + 7

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't convert 'int' object to str implicitly
```

Python is a (fairly) strongly typed language, and it is not possible to add a string to an integer.[2]

A `ValueError`, on the other hand, occurs when the object involved has the correct *type* but an invalid *value*:

```
>>> float('hello')

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: could not convert string to float: 'hello'
```

[2] Unlike in, say, Javascript or PHP, where it seems anything goes.

Table 4.1 Common Python exceptions

Exception	Cause and description
`FileNotFoundError`	Attempting to open a file or directory that does not exist – this exception is a particular type of `OSError`.
`IndexError`	Indexing a sequence (such as a list or string) with a subscript that is out of range.
`KeyError`	Indexing a dictionary with a key that does not exist in that dictionary (see Section 4.2.2).
`NameError`	Referencing a local or global variable name that has not been defined.
`TypeError`	Attempting to use an object of an inappropriate type as an argument to a built-in operation or function.
`ValueError`	Attempting to use an object of the correct type but with an incompatible value as an argument to a built-in operation or function.
`ZeroDivisionError`	Attempting to divide by zero (either explicitly (using '/' or '//') or as part of a modulo operation '`%`').
`SystemExit`	Raised by the `sys.exit` function (see Section 4.4.1) – if not handled, this function causes the Python interpreter to exit.

The `float` built-in does take a string as its argument, so `float('hello')` is not a `TypeError`: the exception is raised because the particular string '`hello`' does not evaluate to a meaningful floating point number. More subtly,

```
>>> int('7.0')

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: '7.0'
```

A string that looks like a `float` cannot be directly cast to `int`: to obtain the result probably intended, use `int(float('7.0'))`.

Table 4.1 provides a list of the more commonly encountered built-in exceptions and their descriptions.

Example E4.2 When an exception is raised but not *handled* (see Section 4.1.3), Python will issue a *traceback report* indicating where in the program flow it occurred. This is particularly useful when an error occurs within nested functions or within imported modules. For example, consider the following short program:[3]

```
# exception-test.py
import math

def func(x):
    def trig(x):
```

[3] Note the use of `f.__name__` to return a string representation of a function's name in this program; for example, `math.sin.__name__` is `'sin'`.

```
        for f in (math.sin, math.cos, math.tan):
            print('{f}({x}) = {res}'.format(f=f.__name__, x=x, res=f(x)))
    def invtrig(x):
        for f in (math.asin, math.acos, math.atan):
❶           print('{f}({x}) = {res}'.format(f=f.__name__, x=x, res=f(x)))
    trig(x)
❷   invtrig(x)

❸ func(1.2)
```

The function `func` passes its argument, `x`, to its two nested functions. The first, `trig`, is unproblematic but the second, `invtrig`, is expected to fail for `x` out of the domain (range of acceptable values) for the inverse trigonometric function, `asin`:

```
sin(1.2) = 0.9320390859672263
cos(1.2) = 0.3623577544766736
tan(1.2) = 2.5721516221263183
Traceback (most recent call last):
  File "exception-test.py", line 14, in <module>
    func(1.2)
  File "exception-test.py", line 12, in func
    invtrig(x)
  File "exception-test.py", line 10, in invtrig
    print('{f}({x}) = {res}'.format(f=f.__name__, x=x, res=f(x)))
ValueError: math domain error
```

Following the traceback backward shows that the `ValueError` exception was raised within `invtrig` (line 10, ❶), which was called from within `func` (line 12, ❷), which was itself called by the `exception-test.py` module (i.e., program) at line 14, ❸.

4.1.3 Handling and raising exceptions

Handling exceptions

Often, a program must manipulate data in a way which might cause an exception to be raised. Assuming such a condition is not to cause the program to exit with an error but to be handled "gracefully" in some sense (an invalid data point ignored, division by a zero value skipped, and so on), there are two approaches to this situation: check the value of the data object before using it, or "handle" any exception that is raised before resuming execution. The Pythonic approach is the latter, summed up in the expression *It is Easier to Ask Forgiveness than to seek Permission* (EAFP).

To catch an exception in a block of code, write the code within a `try:` clause and handle any exceptions raised in an `except:` clause. For example,

```
try:
    y = 1 / x
    print('1 /', x, ' = ',y)
except ZeroDivisionError:
    print('1 / 0 is not defined.')
# ... more statements
```

No check is required: we go ahead and calculate `1/x` and handle the error arising from division by zero if necessary. The program execution continues after the `except` block

whether the `ZeroDivisionError` exception was raised or not. If a different exception is raised (e.g., a `NameError` because `x` is not defined), then this will not be caught – it is an *unhandled exception* and will trigger an error message.

To handle more than one exception in a single `except` block, list them in a tuple (which must be within brackets).

```
try:
    y = 1. / x
    print('1 /', x, ' = ',y)
except (ZeroDivisionError, NameError):
    print('x is zero or undefined!)
# ... more statements
```

To handle each exception separately, use more than one `except` clause:

```
try:
    y = 1. / x
    print('1 /', x, ' = ',y)
except ZeroDivisionError:
    print('1 / 0 is not defined.')
except NameError:
    print('x is not defined')
# ... more statements
```

Warning: You may come across the following type of construction:

```
try:
    [do something]
except:                    # Don't do this!
    pass
```

This will execute the statements in the `try` block and ignore *any* exceptions raised – it is very unwise to do this as it makes code very hard to maintain and debug (errors, whatever their cause, are silently supressed). Always catch specific exceptions and handle them appropriately, allowing any other exceptions to "bubble up" to be handled (or not) by any other `except` clauses.

The `try ... except` statement has two more optional clauses (which must follow any `except` clauses if they are used). Statements in a block following the `finally` keyword are *always* executed, whether an exception was raised or not. Statements in a block following the `else` keyword are executed if an exception was *not* raised (see Example E4.5).

◊ Raising exceptions

Usually an exception is raised by the Python interpreter as a result of some behavior (anticipated or not) by the program. But sometimes it is desirable for a program to raise a particular exception if some condition is met. The `raise` keyword allows a program to force a specific exception and customize the message or other data associated with it. For example,

```
if n % 2:
    raise ValueError('n must be even!')
# statements here may proceed, knowing n is even ...
```

A related keyword, `assert`, evaluates a conditional expression and raises an `AssertionError` exception if that expression is not `True`. This is useful to check that some essential condition holds at a specific point in your program's execution and is often helpful in debugging.

```
>>> assert 2==2         # [silence]: 2==2 is True so nothing happens
>>>
>>> assert 1==2         # Will raise the AssertionError

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AssertionError
```

The syntax `assert` *`expr1`*`,` *`expr2`* passes *`expr2`* (typically an error message) to the `AssertionError`:

```
>>> assert 1==2, 'One does not equal two'

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AssertionError: One does not equal two
```

Python is a dynamically typed language and arguments of any type can be legally passed to a function, even if that function is expecting a particular type. It is sometimes necessary to check that an argument object is of a suitable type before using it, and `assert` could be used to do this.

Example E4.3 The following function returns a string representation of a two-dimensional (2D) or three-dimensional (3D) vector, which must be represented as a `list` or `tuple` containing two or three items.

```
>>> def str_vector(v):
...     assert type(v) is list or type(v) is tuple,\
...             'argument to str_vector must be a list or tuple'
...     assert len(v) in (2,3),\
...             'vector must be 2D or 3D in str_vector'
...     unit_vectors = ['i','j','k']
...     s = []
...     for i, component in enumerate(v):
...         s.append('{}{}'.format(component, unit_vectors[i]))
❶ ...     return '+'.join(s).replace('+-', '-')
```

❶ `replace('+-', '-')` here converts, for example, `'4i+-3j'` into `'4i-3j'`.

Example E4.4 As another example, suppose you have a function that calculates the vector (cross) product of two vectors represented as `list` objects. This product is only defined for three-dimensional vectors, so calling it with lists of any other length is an error.

```
>>> def cross_product(a, b):
...     assert len(a) == len(b) == 3, 'Vectors a, b must be three-dimensional'
...     return [a[1]*b[2] - a[2]*b[1],
...             a[2]*b[0] - a[0]*b[2],
...             a[0]*b[1] - a[1]*b[0]]
...
>>> cross_product([1, 2, -1], [2, 0, -1, 3])    # Oops

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in cross_product
AssertionError: Vectors a, b must be three-dimensional
>>> cross_product([1, 2, -1], [2, 0, -1])
[-2, -1, -4]
```

Example E4.5 The following code gives an example of the use of a `try ... except ... else ... finally` clause:

```
# try-except-else-finally.py

def process_file(filename):
    try:
        fi = open(filename, 'r')
    except IOError:
        print('Oops: couldn\'t open {} for reading'.format(filename))
        return
    else:
❶       lines = fi.readlines()
        print('{} has {} lines.'.format(filename, len(lines)))
        fi.close()
    finally:
❷       print('   Done with file {}'.format(filename))

    print('The first line of {} is:\n{}'.format(filename, lines[0]))
    # further processing of the lines ...
    return

process_file('sonnet0.txt')
process_file('sonnet18.txt')
```

❶ Within the `else` block, the contents of the file are only read if the file was successfully opened.

❷ Within the `finally` block, '`Done with file` *`filename`*' is printed whether the file was successfully opened or not.

Assuming that the file `sonnet0.txt` does not exist but that `sonnet18.txt` does, running this program prints:

```
Oops: couldn't open sonnet0.txt for reading
   Done with file sonnet0.txt
sonnet18.txt has 14 lines.
   Done with file sonnet18.txt
The first line of sonnet18.txt is:
Shall I compare thee to a summer's day?
```

4.1.4 Exercises

Questions

Q4.1.1 What is the point of `else`? Why not put statements in this block inside the original `try` block?

Q4.1.2 What is the point of the `finally` clause? Why not put any statements you want executed after the `try` block (regardless of whether or not an exception has been raised) after the entire `try ... except` clause?

Hint: see what happens if you modify Example E4.5 to put the statements in the `finally` clause after the `try` block.

Problems

P4.1.1 Write a program to read in the data from the file `swallow-speeds.txt` (available at scipython.com/ex/ada) and use it to calculate the average air-speed velocity of an (unladen) African swallow. Use exceptions to handle the processing of lines that do not contain valid data points.

P4.1.2 Adapt the function of Example E4.3, which returns a vector in the following form:

```
>>> print(str_vector([-2, 3.5])
-2i + 3.5j
>>> print(str_vector((4, 0.5, -2))
4i + 0.5j - 2k
```

to raise an exception if any element in the vector array does not represent a real number.

P4.1.3 Python follows the convention of many computer languages in choosing to define $0^0 = 1$. Write a function, `powr(a, b)`, which behaves the same as the Python expression `a**b` (or, for that matter, `math.pow(a,b)`) but raises a `ValueError` if `a` and `b` are both zero.

4.2 Python objects III: dictionaries and sets

A *dictionary* in Python is a type of "associative array" (also known as a "hash" in some languages). A dictionary can contain any objects as its *values*, but unlike sequences such as lists and tuples, in which the items are indexed by an integer starting at 0, each item in a dictionary is indexed by a unique *key*, which may be any *immutable* object.[4] The dictionary therefore exists as a number of *key-value* pairs, which do not have any particular order. Dictionaries themselves are *mutable* objects.

[4] Actually, dictionary keys can be any *hashable* object: a hashable object in Python is one with a special method for generating a particular integer from any instance of that object; the idea is that instances (which may be large and complex) that compare as equal should have hash numbers that also compare as equal so they can be rapidly looked up in a *hash table*. This is important for some data structures and for optimizing the speed of algorithms involving their objects.

4.2.1 Defining and indexing a dictionary

An dictionary can be defined by giving *key: value* pairs between braces:

```
>>> height = {'Burj Khalifa': 828., 'One World Trade Center': 541.3,
              'Mercury City Tower': -1., 'Q1': 323.,
              'Carlton Centre': 223., 'Gran Torre Santiago': 300.,
              'Mercury City Tower': 339.}
>>> height
{'Q1': 323.0,
 'Burj Khalifa': 828.0,
 'Carlton Centre': 223.0,
 'One World Trade Center': 541.3,
 'Mercury City Tower': 339.0,
 'Gran Torre Santiago': 300.0}
```

The command `print(height)` will return the dictionary in the same format (between braces), but in no particular order. If the same key is attached to different values (as `'Mercury City Tower'` is here), only the most recent value survives: the keys in a dictionary are unique.

An individual item can be retreived by indexing it with its key, either as a literal (`'Q1'`) or with a variable equal to the key:

```
>>> height['One World Trade Center']
541.3
>>> building = 'Carlton Centre'
>>> height[building]
223.0
```

Items in a dictionary can also be *assigned* by indexing it in this way:

```
height['Empire State Building'] = 381.
height['The Shard'] = 306.
```

An alternative way of defining a dictionary is to pass a sequence of `(key, value)` pairs to the `dict` constructor. If the keys are simple strings (of the sort that could be used as variable names), the pairs can also be specified as keyword arguments to this constructor:

```
>>> ordinal = dict([(1, 'First'), (2, 'Second'), (3, 'Third')])
>>> mass = dict(Mercury=3.301e23, Venus=4.867e24, Earth=5.972e24)
>>> ordinal[2]      # NB 2 here is a key, not an index
'Second'
>>> mass['Earth']
5.972e+24
```

A `for`-loop iteration over a dictionary returns the dictionary *keys* (in no particular order):

```
>>> for c in ordinal:
...     print(c, ordinal[c])
...
3 Third
1 First
2 Second
```

Example E4.6 A simple dictionary of roman numerals:

```
>>> numerals = {'one':'I', 'two':'II', 'three':'III', 'four':'IV', 'five':'V',
                'six':'VI', 'seven':'VII', 'eight':'VIII',
                1: 'I', 2: 'II', 3: 'III', 4:'IV', 5: 'V', 6:'VI', 7:'VII',
                8:'VIII'}
>>> for i in ['three', 'four', 'five', 'six']:
...     print(numerals[i], end=' ')
...
III IV V VI
>>> for i in range(8,0,-1):
...     print(numerals[i], end=' ')
VIII VII VI V IV III II I
```

Note that even though the keys are stored in an arbitrary order, the dictionary can be indexed in any order. Note also that although the dictionary *keys* must be unique, the dictionary *values* need not be.

4.2.2 Dictionary methods

`get()`

Indexing a dictionary with a key that does not exist is an error:

```
>>> mass['Pluto']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'Pluto'
```

However, the useful method `get()` can be used to retrieve the value, given a key if it exists, or some default value if it does not. If no default is specified, then `None` is returned. For example,

```
>>> print(mass.get('Pluto'))
None
>>> mass.get('Pluto', -1)
-1
```

`keys`, `values` and `items`

The three methods, `keys`, `values` and `items`, return respectively, a dictionary's keys, values and key-value pairs (as tuples). In previous versions of Python, each of these were returned in a list, but for most purposes this is wasteful of memory: calling `keys`, for example, required all of the dictionary's keys to to be copied as a list, which in most cases was simply iterated over. That is, storing a whole new copy of the dictionary's keys is not usually necessary. Python 3 solves this by returning an iterable object, which accesses the dictionary's keys one by one, without copying them to a list. This is faster and saves memory (important for very large dictionaries). For example,

```
>>> planets = mass.keys()
>>> print(planets)
dict_keys(['Venus', 'Mercury', 'Earth'])
>>> for planet in planets:
...     print(planet, mass[planet])
```

```
...
Venus 4.867e+24
Mercury 3.301e+23
Earth 5.972e+24
```

A `dict_keys` object can be iterated over any number of times, but it is not a list and cannot be indexed or assigned:

```
>>> planets = mass.keys()
>>> planets[0]

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'dict_keys' object is not subscriptable
```

If you really do want a list of the dictionary's keys, simply pass the `dict_keys` object to the `list` constructor (which takes any kind of sequence and makes a `list` out of it):

```
>>> planet_list = list(mass.keys())
>>> planet_list[0]
'Venus'
❶ >>> planet_list[1] = 'Jupiter'
>>> planet_list
['Venus', 'Jupiter', 'Earth']
```

❶ This last assignment only changes the `planet_list` list; it doesn't alter the original dictionary's keys.

Similar methods exist for retrieving a dictionary's values and items (key-value pairs): the objects returned are `dict_values` and `dict_items`.

For example,

```
>>> mass.items()
dict_items([('Venus', 4.867e+24), ('Mercury', 3.301e+23), ('Earth', 5.972e+24)])
>>> mass.values()
dict_values([4.867e+24, 3.301e+23, 5.972e+24])
>>> for planet_data in mass.items():
...     print(planet_data)
...
('Venus', 4.867e+24)
('Mercury', 3.301e+23)
('Earth', 5.972e+24)
```

Example E4.7 A Python dictionary can act as a kind of simple database. The following code stores some information about some astronomical objects in a dictionary of tuples, keyed by the object name, and manipulates them to produce a list of planet densities.

Listing 4.1 Astronomical data

```
# eg4-astrodict.py
import math

# Mass (in kg) and radius (in km) for some astronomical bodies
body = {'Sun': (1.988e30, 6.955e5),
        'Mercury': (3.301e23, 2440.),
        'Venus': (4.867e+24, 6052.),
```

```
        'Earth': (5.972e24, 6371.),
        'Mars': (6.417e23, 3390.),
        'Jupiter': (1.899e27, 69911.),
        'Saturn': (5.685e26, 58232.),
        'Uranus': (8.682e25, 25362.),
        'Neptune': (1.024e26, 24622.)
       }

planets = list(body.keys())
# The sun isn't a planet!
planets.remove('Sun')

def calc_density(m, r):
    """ Returns the density of a sphere with mass m and radius r. """
    return m / (4/3 * math.pi * r**3)

rho = {}
for planet in planets:
    m, r = body[planet]
    # calculate the density in g/cm3
    rho[planet] = calc_density(m*1000, r*1.e5)

for planet, density in sorted(rho.items()):   # ❶
    print('The density of {0} is {1:3.2f} g/cm3'.format(planet, density))
```

❶ `sorted(rho.items())` returns a `list` of the `rho` dictionary's key-value pairs, sorted by key.

The output is

```
The density of Earth is 5.51 g/cm3
The density of Jupiter is 1.33 g/cm3
The density of Mars is 3.93 g/cm3
The density of Mercury is 5.42 g/cm3
The density of Neptune is 1.64 g/cm3
The density of Saturn is 0.69 g/cm3
The density of Uranus is 1.27 g/cm3
The density of Venus is 5.24 g/cm3
```

◊ Keyword arguments

In Section 2.7, we discussed the syntax for passing arguments to functions. In that description, it was assumed that the function would always know what arguments could be passed to it and these were listed in the function definition. For example,

```
def func(a, b, c):
```

Python provides a couple of useful features for handling the case where it is not necessarily known what arguments a function will receive. Including `*args` (after any "formally defined" arguments) places any additional positional argument into a `tuple`, `args`, as illustrated by the following code:

```
>>> def func(a, b, *args):
...     print(args)
...
```

```
>>> func(1, 2, 3, 4, 'msg')
(3, 4, 'msg')
```

That is, inside `func`, in addition to the formal arguments `a=1` and `b=2`, the arguments `3`, `4` and `'msg'` are available as the items of the `tuple args`. This `tuple` can be arbitrarily long. Python's own `print` built-in function works in this way: it takes an arbitrary number of arguments to output as a string, followed by some optional keyword arguments:

```
def print(*args, sep=' ', end='\n', file=None):
```

It is also possible to collect arbitrary *keyword arguments* (see Section 2.7.2) to a function inside a dictionary by using the `**kwargs` syntax in the function definition. Python takes any keyword arguments not specified in the function definition and packs them into the dictionary `kwargs`. For example,

```
>>> def func(a, b, **kwargs):
...     for k in kwargs:
...         print(k, '=', kwargs[k])
...
>>> func(1, b=2, c=3, d=4, s='msg')
d = 4
s = msg
c = 3
```

One can also use `*args` and `**kwargs` when *calling* a function, which can be convenient, for example, with functions that take a large number of arguments:

```
>>> def func(a, b, c, x, y, z):
...     print(a, b, c)
...     print(x, y, z)
...
>>> args = [1,2,3]
>>> kwargs = {'x': 4, 'y': 5, 'z': 'msg'}
>>> func(*args, **kwargs)
1 2 3
4 5 msg
```

4.2.3 Sets

A `set` is an *unordered* collection of *unique* items. As with dictionary keys, elements of a set must be hashable objects. A set is useful for removing duplicates from a sequence and for determining the union, intersection and difference between two collections. Because they are unordered, `set` objects cannot be indexed or sliced, but they can be iterated over, tested for membership, and they support the `len` built-in. A `set` is created by listing its elements between braces ({. . .}) or by passing an iterable to the `set()` constructor:

```
>>> s = set([1, 1, 4, 3, 2, 2, 3, 4, 1, 3, 'surprise!'])
>>> s
{1, 2, 'surprise!', 3, 4}
>>> len(s)                  # cardinality of the set
5
```

```
>>> 2 in s, 6 not in s       # membership, nonmembership
(True, True)
>>> for item in s:
...     print(item)
...
1
2
surprise!
3
4
```

The `set` method `add` is used to add elements to the set. To remove elements there are several methods: `remove` removes a specified element but raises a `KeyError` exception if the element is not present in the set; `discard()` does the same but does not raise an error in this case. Both methods take (as a single argument) the element to be removed. `pop` (with no argument) removes an *arbitrary* element from the set and `clear` removes *all* elements from the set:

```
>>> s = {2,-2,0}
>>> s.add(1)
>>> s.add(-1)
>>> s.add(1.0)            ❶
>>> s
{0, 1, 2, -1, -2}
>>> s.remove(1)
>>> s
{0, 2, -1, -2}
>>> s.discard(3)     # OK - does nothing
>>> s
{0, 2, -1, -2}
>>> s.pop()
0                    # (for example)
>>> s
{2, -1, -2}
>>> s.clear()
set()                # the empty set
```

❶ This statement will not add a new member to the set, even though the existing `1` is an integer and the item we're adding is a `float`. The test `1 == 1.0` is `True`, so `1.0` is considered to be already in the set.

`set` objects have a wide range of methods corresponding to the properties of mathematical sets; the most useful are illustrated in Table 4.2, which uses the following terms from set theory:

- The *cardinality* of a set, $|A|$, is the number of elements it contains.
- Two sets are *equal* if they both contain the same elements.
- Set A is a *subset* of set B ($A \subseteq B$) if all the elements of A are also elements of B; set B is said to be a *superset* of set A.
- Set A is a *proper subset* of B ($A \subset B$) if it is a subset of B but not equal to B; in this case, set B is said to be a *proper superset* of A.
- The *union* of two sets ($A \cup B$) is the set of all elements from both of them.
- The *intersection* of two sets ($A \cap B$) is the set of all elements they have in common.

Table 4.2 `set` methods

Method	Description
`isdisjoint(other)`	Is `set` disjoint with `other`?
`issubset(other)`, `set <= other`	Is `set` a subset of `other`?
`set < other`	Is `set` a proper subset of `other`?
`issuperset(other)`, `set >= other`	Is `set` a superset of `other`?
`set > other`	Is `set` a proper superset of `other`?
`union(other)` `set | other | ...`	The union of `set` and `other`(s)
`intersection(other)`, `set & other & ...`	The intersection of `set` and `other`(s)
`difference(other)`, `set - other - ...`	The difference of `set` and `other`(s)
`symmetric_difference(other)`, `set ^ other ^ ...`	The symmetric difference of `set` and `other`(s)

- The *difference* of set A and set B ($A \backslash B$) is the set of elements in A that are not in B.
- The *symmetric difference* of two sets, $A \triangle B$, is the set of elements in either but not in both.
- Two sets are said to be *disjoint* if they have no elements in common.

There are two forms for most `set` expressions: the operator-like syntax requires all arguments to be `set` objects, whereas explicit method calls will convert any iterable argument into a `set`.

```
>>> A = set((1, 2, 3))
>>> B = set((1, 2, 3, 4))
>>> A <= B
True
>>> A.issubset((1, 2, 3, 4))  # OK: (1, 2, 3, 4) is turned into a set
True
```

Some more examples:

```
>>> C, D = set((3, 4, 5, 6)), set((7, 8, 9))
>>> B | C               # union
{1, 2, 3, 4, 5, 6}
>>> A | C | D           # union of three sets
{1, 2, 3, 4, 5, 6, 7, 8, 9}
>>> A & C               # intersection
{3}
>>> C & D
set()                   # the empty set
>>> C.isdisjoint(D)
True
>>> B - C               # difference
{1, 2}
>>> B ^ C               # symmetric difference
{1, 2, 5, 6}
```

◊ frozensets

`sets` are mutable objects (items can be added to and removed from a `set`); because of this they are *unhashable* and so cannot be used as dictionary keys or as members of other sets.

```
>>> a = set((1,2,3))
>>> b = set(('q', (1,2), a))

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'set'
>>>
```

(In the same way, `lists` cannot be dictionary keys or set members.) There is, however, a `frozenset` object which is a kind of immutable (and hashable) `set`.[5] `frozensets` are fixed, unordered collections of unique objects and *can* be used as dictionary keys and set members.

```
>>> a = frozenset((1,2,3))
>>> b = set(('q', (1,2), a))     # OK: the frozenset a is hashable
>>> b.add(4)                      # OK: b is a regular set
>>> a.add(4)                      # Not OK: frozensets are immutable

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'frozenset' object has no attribute 'add'
```

Example E4.8 A *Mersenne prime*, M_i, is a prime number of the form $M_i = 2^i - 1$. The set of Mersenne primes less than n may be thought of as the intersection of the set of all primes less than n, P_n, with the set, A_n, of integers satisfying $2^i - 1 < n$.

The following program returns a list of the Mersenne primes less than 1000000.

Listing 4.2 The Mersenne primes

```
import math

def primes(n):
    """ Return a list of the prime numbers <= n. """

    sieve = [True] * (n // 2)
    for i in range(3, int(math.sqrt(n))+1, 2):
        if sieve[i//2]:
            sieve[i*i//2::i] = [False] * ((n - i*i - 1) // (2*i) +1)
    return [2] + [2*i+1 for i in range(1, n // 2) if sieve[i]]

n = 1000000

❶ P = set(primes(n))
```

[5] In a sense, they are to `sets` what `tuples` are to `lists`.

```
   # A list of integers 2^i-1 <= n
   A = []
❷  for i in range(2, int(math.log(n+1, 2))+1):
       A.append(2**i - 1)

   # The set of Mersenne primes as the intersection of P and A
   M = P.intersection(A)

   # Output as a sorted list of M
❸  print(sorted(list(M)))
```

The prime numbers are produced in a list by the function `primes`, which implements an optimized version of the Sieve of Eratosthenes algorithm (see Exercise P2.5.8); this is converted into the `set`, `P` (❶). We can take the intersection of this set with any iterable object using the `intersection` method, so there is no need to explicitly convert our second list of integers, `A`, (❷) into a `set`.

❸ Finally, the set of Mersenne primes we create, `M`, is an unordered collection, so for output purposes we convert it into a sorted `list`.

For $n = 1000000$, This output is

```
[3, 7, 31, 127, 8191, 131071, 524287]
```

4.2.4 Exercises

Questions

Q4.2.1 Write a one-line Python program to determine if a string is a *pangram* (a string that contains each letter of the alphabet at least once).

Q4.2.2 Write a function, using `set` objects, to remove duplicates from an ordered `list`. For example,

```
>>> remove_dupes([1,1,2,3,4,4,4,5,7,8,8,9])
[1, 2, 3, 4, 5, 7, 8, 9]
```

Q4.2.3 Predict and explain the effect of the following statements:

```
>>> set('hellohellohello')
>>> set(['hellohellohello'])
>>> set(('hellohellohello'))
>>> set(('hellohellohello',))
>>> set(('hello', 'hello', 'hello'))
>>> set(('hello', ('hello', 'hello')))
>>> set(('hello', ['hello', 'hello']))
```

Q4.2.4 If `frozenset` objects are *immutable*, how is this possible?

```
>>> a = frozenset((1,2,3))
>>> a |= {2,3,4,5}
>>> print(a)
frozenset([1, 2, 3, 4, 5])
```

Table 4.3 Resistor color codes

Color	Abbreviation	Significant figures	Multiplier	Tolerance
Black	`bk`	0	1	–
Brown	`br`	1	10	±1%
Red	`rd`	2	10^2	±2%
Orange	`or`	3	10^3	–
Yellow	`yl`	4	10^4	±5%
Green	`gr`	5	10^5	±0.5%
Blue	`bl`	6	10^6	±0.25%
Violet	`vi`	7	10^7	±0.1%
Gray	`gy`	8	10^8	±0.05%
White	`wh`	9	10^9	–
Gold	`au`	–	–	±5%
Silver	`ag`	–	–	±10%
None	`--`	–	–	±20%

Problems

P4.2.1 The values and tolerances of older resistors are identified by four colored bands: the first two indicate the first two significant figures of the resistance in ohms, the third denotes a decimal multiplier (number of zeros), and the fourth indicates the tolerance. The colors and their meanings for each band are listed in Table 4.3.

For example, a resistor with colored bands violet, yellow, red, green has value $74 \times 10^2 = 7400\ \Omega$ and tolerance $\pm 0.5\%$.

Write a program that defines a function to translate a list of four color abbreviations into a resistance value and a tolerance. For example,

```
In [x]: print(get_resistor_value(['vi', 'yl', 'rd', 'gr']))
Out[x]: (7400, 0.5)
```

P4.2.2 The novel *Moby-Dick* is out of copyright and can be downloaded as a text file from the Project Gutenberg website at www.gutenberg.org/2/7/0/2701/. Write a program to output the 100 words most frequently used in the book by storing a count of each word encountered in a dictionary.

Hint: use Python's string methods to strip out any punctuation. It suffices to replace any instances of the following characters with the empty string: `!?":;,()'.*[]`. When you have a dictionary with words as the keys and the corresponding word counts as the values, create a list of (*count*, *word*) tuples and sort it.

Bonus exercise: compare the frequencies of the top 2000 words in *Moby-Dick* with the prediction of *Zipf's Law*:

$$\log f(w) = \log C - a \log r(w),$$

where $f(w)$ is the number of occurrences of word w, $r(w)$ is the corresponding *rank* (1 = most common, 2 = second most common, etc.) and C and a are constants. In the

traditional formulation of the law, $C = \log f(w_1)$ and $a = 1$, where w_1 is the most common word, such that $r(w_1) = 1$.

P4.2.3 *Reverse notation* (RPN) (or *postfix* notation) is a notation for mathematical expressions in which each operator follows all of its operands (in contrast to the more familiar *infix* notation, in which the operator appears *between* the operands it acts on). For example, the infix expression `5+6` is written in RPN as `5 6 +`. The advantage of this approach is that parentheses are not necessary: to evaluate `(3+7) / 2`, it may be written as `3 7 + 2 /`. An RPN expression is evaluated left to right with the intermediate values pushed onto a *stack* – a last-in, first-out list of values – and retrieved (popped) from the stack when needed by an operator. Thus, the expression `3 7 + 2 /` proceeds with `3` and then `7` pushed to the stack (with 7 on top). The next token is `+`, so the values are retrieved, added, and the result, `10` pushed onto the (now empty) stack. Next, `2` is pushed to the stack. The final token `/` pops the two values, `10` and `2` from the stack, and divides them to give the result, `5`.

Write a program to evaluate an RPN expression consisting of space-delimited tokens (the operators `+ - * / **` and numbers).

Hint: parse the expression into a list of string tokens and iterate over it, converting and pushing the numbers to the stack (which may be implemented by appending to a `list`). Define functions to carry out the operations by retrieving values from the stack with `pop`. Note that Python does not provide a `switch...case` syntax, but these function objects can be the values in a dictionary with the operator tokens as the keys.

P4.2.4 Use the dictionary of Morse code symbols available from scipython.com/ex/adb to write a program that can translate a message to and from Morse code, using spaces to delimit individual Morse code "letters" and slashes ('/') to delimit words. For example, `'PYTHON 3'` becomes `'.-. -.- - .... -- -. / ...-'`

P4.2.5 The file `shark-species.txt`, available at scipython.com/ex/adc, contains a list of extant shark species arranged in a hierachy by order, family, genus and species (with the species given as *binomial name : common name*). Read the file into a data structure of nested dictionaries, which can be accessed as follows:

```
>>> sharks['Lamniformes']['Lamnidae']['Carcharodon']['C. carcharias']
Great white shark
```

4.3 Pythonic idioms: "syntactic sugar"

Many computer languages provide syntax to make common tasks easier and clearer to code. Such *syntactic sugar* consists of constructs that could be removed from the language without affecting the language's functionality. We have already seen one example in so-called *augmented assignment*: `a += 1` is equivalent to `a = a + 1`. Another

example is negative indexing of sequences: `b[-1]` is equivalent to and more convenient than `b[len(b)-1]`.

4.3.1 Comparison and assignment shortcuts

If more than one variable is to be assigned to the same object, the shortcut

```
x = y = z = -1
```

may be used. Note that if *mutable* objects are assigned this way, the variable names will all refer to the same object, not to distinct copies of it (recall Section 2.4.1).

Similarly, as was shown in Section 2.4.2, multiple assignments to different objects can be acheived in a single line by *tuple unpacking*:

```
a, b, c = x + 1, 'hello', -4.5
```

The tuple on the right-hand side of this expression (parentheses are optional in this case) is unpacked in the assignment to the variable names on the left-hand side. This single line is thus equivalent to the three lines

```
a = x + 1
b = 'hello'
c = -4.5
```

In expressions such as these the right-hand side is evaluated first and then assigned to the left-hand side. As we have seen in Section 2.4.2, this provides a very useful way of swapping the value of two variables without the need for a temporary variable:

```
a, b = b, a
```

Comparisons may also be chained together in a natural way:

```
if a == b == 3:
    print('a and b both equal 3')
if -1 < x < 1:
    print('x is between -1 and 1')
```

Python supports *conditional assignment*: a variable name can be set to one value or another depending on the outcome of an `if ... else` expression on the same line as the assignment. For example,

```
y = math.sin(x)/x if x else 1
```

Short examples such as this one, in which the potential division by zero is avoided (recall that `0` evaluates to `False`) are benign enough, but the idiom should be avoided for anything more complex in favor of a more explicit construct such as

```
try:
    y = math.sin(x)/x
except ZeroDivisionError:
    y = 1
```

4.3.2 List comprehension

A list comprehension in Python is a construct for creating a list based on another iterable object in a single line of code. For example, given a list of numbers, `xlist`, a list of the squares of those numbers may be generated as follows:

```
>>> xlist = [1, 2, 3, 4, 5, 6]
>>> x2list = [x**2 for x in xlist]
>>> x2list
[1, 4, 9, 16, 25, 36]
```

This is a faster and syntactically nicer way of creating the same list with a block of code within a `for` loop:

```
>>> x2list = []
>>> for x in xlist:
...     x2list.append(x**2)
```

List comprehensions can also contain conditional statements:

```
>>> x2list = [x**2 for x in xlist if x % 2]
>>> x2list
[1, 9, 25]
```

Here, `x` gets fed to the `x**2` expression to be entered into the `x2list` under construction only if `x % 2` evaluates to `True` (i.e., if `x` is *odd*). This is an example of a *filter* (a single `if` conditional expression). If you require a more complex *mapping* of values in the original sequence to values in the constructed list, the `if .. else` expression must appear before the `for` loop:

```
>>> [x**2 if x % 2 else x**3 for x in xlist]
[1, 8, 9, 64, 25, 216]
```

This comprehension squares the odd integers and cubes the even integers in `xlist`.

Of course, the sequence used to construct the list does not have to be another list. For example, strings, tuples and `range` objects are all iterable and can be used in list comprehensions:

```
>>> [x**3 for x in range(1,10)]
[1, 8, 27, 64, 125, 216, 343, 512, 729]
>>> [w.upper() for w in 'abc xyz']
['A', 'B', 'C', ' ', 'X', 'Y', 'Z']
```

Finally, list comprehensions can be nested. For example, the following code flattens a list of lists:

```
>>> vlist = [[1,2,3], [4,5,6], [7,8,9]]
>>> [c for v in vlist for c in v]
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Here, the first loop produces the inner lists, one by one, as `v`, and each inner list `v` is iterated over as `c` to be added to the list being created.

Example E4.9 Consider a 3×3 matrix represented by a list of lists:

```
M = [[1,2,3],
     [4,5,6],
     [7,8,9]]
```

Without using list comprehension, the transpose of this matrix could be built up by looping over the rows and columns:

```
MT = [[0,0,0], [0,0,0], [0,0,0]]
for ir in range(3):
    for ic in range(3):
        MT[ic][ir] = M[ir][ic]
```

With one list comprehension, the transpose can be constructed as

```
MT = []
for i in range(3):
    MT.append([row[i] for row in M])
```

where rows of the transposed matrix are built from the columns (indexed with `i=0,1,2`) of each row in turn from `M`). The outer loop here can be expressed as a list comprehension of its own:

```
MT = [[row[i] for row in M] for i in range(3)]
```

Note, however, that NumPy provides a much easier way to manipulate matrices (see Section 6.6).

4.3.3 `lambda` functions

A `lambda` function in Python is a type of simple *anonymous function*. The executable body of a `lambda` function must be an *expression* and not a *statement*; that is, it may *not* contain, for example, loop blocks, conditionals or `print` statements. `lambda` functions provide limited support for a programming paradigm known as *functional programming*.[6] The simplest application of a `lambda` function differs little from the way a regular function `def` would be used:

```
>>> f = lambda x: x**2 - 3*x + 2
>>> print(f(4.))
6.0
```

The argument is passed to `x` and the result of the function specified in the `lambda` definition after the colon is passed back to the caller. To pass more than one argument to a `lambda` function, pass a tuple (without parentheses):

```
>>> f = lambda x,y: x**2 + 2*x*y + y**2
>>> f(2., 3.)
25.0
```

[6] Functional programming is a style of programming in which computation is achieved through the evaluation of mathematical functions with minimal reference to variables defining the *state* of the program.

In these examples, not too much is gained by using a `lambda` function, and the functions defined are not all that anonymous either (because they've been bound to the variable name `f`). A more useful application is in creating a list of functions, as in the following example.

Example E4.10 Functions are objects (like everything else in Python) and so can be stored in lists. Without using `lambda` we would have to define named functions (using `def`) before constructing the list:

```
def const(x):
    return 1.
def lin(x):
    return x
def square(x):
    return x**2
def cube(x):
    return x**3
flist = [const, lin, square, cube]
```

Then `flist[3](5)` returns 125, since `flist[3]` is the function `cube`, and is called with the argument 5.

The value of using `lambda` expressions as anonymous functions is that if these functions do not need to be named if they are just to be stored in a list and so can be defined as items "inline" with the list construction:

```
>>> flist = [lambda x: 1,
...          lambda x: x,
...          lambda x: x**2,
...          lambda x: x**3]
>>> flist[3](5)    # flist[3] is x**3
125
>>> flist[2](4)    # flist[2] is x**2
16
```

Example E4.11 The `sorted` built-in and `sort` list method can order lists based on the returned value of a function called on each element prior to making comparisons. This function is passed as the `key` argument. For example, sorting a list of strings is case sensitive by default:

```
>>> sorted('Nobody expects the Spanish Inquisition'.split())
['Inquisition', 'Nobody', 'Spanish', 'expects', 'the']
```

We can make the sorting case insensitive, however, by passing each word to the `str.lower` method:

```
>>> sorted('Nobody expects the Spanish Inquisition'.split(), key=str.lower)
['expects', 'Inquisition', 'Nobody', 'Spanish', 'the']
```

(Of course, `key=str.upper` would work just as well.) Note that the list elements themselves are not altered: they are being ordered based on a lowercase version of

themselves. We do not use parentheses here, as in `str.lower()`, because we are passing the *function itself* to the `key` argument, not calling it directly.

It is typical to use `lambda` expressions to provide simple anonymous functions for this purpose. For example, to sort a list of atoms as (element symbol, atomic number) tuples in order of atomic number (the *second* item in each tuple):

```
>>> halogens = [('At', 85), ('Br', 35), ('Cl', 17), ('F', 9), ('I', 53)]
>>> sorted(halogens, key=lambda e: e[1])
[('F', 9), ('Cl', 17), ('Br', 35), ('I', 53), ('At', 85)]
```

Here, the sorting algorithm calls the function specified by `key` on each tuple item to decide where it belongs in the sorted list. Our anonymous function simply returns the second element of each tuple, and so sorting is by atomic number.

4.3.4 The `with` statement

The `with` statement creates a block of code that is executed within a certain *context*. A context is defined by a *context manager* that provides a pair of methods describing how to enter and leave the context. User-defined contexts are generally used only in advanced code and can be quite complex, but a common basic example of a built-in context manager involves file input / output. Here, the context is entered by opening the file. Within the context block, the file is read from or written to, and finally the file is closed on exiting the context. The `file` object is a context manager that is returned by the `open()` method. It defines an exit method which simply closes the file (if it was opened successfully), so that this does not need to be done explicitly. To open a file within a context, use

```
with open('filename') as f:
    # process the file in some way, for example:
    lines = f.readlines()
```

The reason for doing this is that you can be sure that the file will be closed after the `with` block, even if something goes wrong in this block: the context manager handles the code you would otherwise have to write to catch such runtime errors.

4.3.5 Generators

Generators are a powerful feature of the Python language; they allow one to declare a function that behaves like an iterable object. That is, a function that can be used in a `for` loop and that will yield its values, in turn, on demand. This is often more efficient than calculating and storing all of the values that will be iterated over (particularly if there will be a very large number of them). A generator function looks just like a regular Python function, but instead of exiting with a `return` value, it contains a `yield` statement which returns a value each time it is required to by the iteration.

A very simple example should make this clearer. Let's define a generator, `count`, to count to `n`:

```
>>> def count(n):
...     i=0
...     while i < n:
...         i += 1
...         yield i
...
>>> for j in count(5):
...     print(j)
...
1
2
3
4
5
```

Note that we can't simply call our generator like a regular function:

```
>>> count(5)
<generator object count at 0x102d8e6e0>
```

The generator `count` is expecting to be called as part of an loop (here, the `for` loop) and on each iteration it `yield`s its result and stores its state (the value of `i` reached) until the loop next calls upon it.

In fact, we have been using generators already because the familiar `range` built-in function is, in Python 3, a type of generator object.

There is a *generator comprehension* syntax similar to list comprehension (use round brackets instead of square brackets):

```
>>> squares = (x**2 for x in range(5))
>>> for square in squares:
...     print(square)
...
0
1
4
9
16
```

However, once we have "exhausted" our generator comprehension defined in this way, we cannot iterate over it again without redefining it. If we try:

```
>>> for square in squares:
...     print(square)
...
>>>
```

we get nothing as we have already reached the end of the `squares` generator.

To obtain a list or tuple of a generator's values, simply pass it to `list` or `tuple`, as shown in the following example.

Example E4.12 This function defines a generator for the *triangular numbers*, $T_n = \sum_{k=1}^{n} k = 1 + 2 + 3 + \cdots + n$, for $n = 0, 1, 2, \cdots$: that is, $T_n = 0, 1, 3, 6, 10, \cdots$.

```
>>> def triangular_numbers(n):
...     i, t = 1, 0
```

```
...     while i <= n:
...         yield t
...         t += i
...         i += 1
...
>>> list(triangular_numbers(15))
[0, 1, 3, 6, 10, 15, 21, 28, 36, 45, 55, 66, 78, 91, 105]
```

Note that the statements after the `yield` statement are executed each time `triangular_numbers` resumes. The call to `triangular_numbers(15)` returns an iterator that feeds these numbers into `list` to generate a list of its values.

4.3.6 ◊ `map`

The built-in function `map` returns an iterator that applies a given function to every item of a provided sequence, yielding the results as a generator would.[7] For example, one way to sum a list of lists is to map the `sum` built-in to it:

```
>>> mylists = [[1,2,3], [10, 20, 30], [25, 75, 100]]
>>> list(map(sum, mylists))
[6, 60, 200]
```

(We have to cast explicitly back to a `list` because `map` returns a generator-like object.) This statement is equivalent to the list comprehension:

```
>>> [sum(l) for l in mylists]
[6, 60, 200]
```

`map` is occasionally useful but has the potential to create very obscure code, and list or generator comprehensions are generally to be preferred. The same applies to the `filter` built-in, which constructs an iterator from the elements of a given sequence for which a provided function returns `True`. In the following example, the odd integers less than 10 are generated: this function returns `x % 2`, and this expression evaluates to 0, equivalent to `False` if `x` is even:

```
>>> list(filter(lambda x: x%2, range(10)))
[1, 3, 5, 7, 9]
```

Again, the list comprehension is more expressive:

```
>>> [x for x in range(10) if x % 2]
[1, 3, 5, 7, 9]
```

4.3.7 Exercises

Questions

Q4.3.1 Rewrite the list of `lambda` functions created in Example E4.10 using a single list comprehension.

[7] Constructs such as `map` are frequently used in functional programming.

Q4.3.2 What does the following code do and how does it work?

```
>>> nmax = 5
>>> x = [1]
>>> for n in range(1,nmax+2):
...     print(x)
...     x = [([0]+x)[i] + (x+[0])[i] for i in range(n+1)]
...
```

Q4.3.3 Consider the lists

```
>>> a = ['A', 'B', 'C', 'D', 'E', 'F', 'G']
>>> b = [4, 2, 6, 1, 5, 0, 3]
```

Predict and explain the output of the following statements:

a. `[a[x] for x in b]`
b. `[a[x] for x in sorted(b)]`
c. `[a[b[x]] for x in b]`
d. `[x for (y,x) in sorted(zip(b,a))]`

Q4.3.4 Dictionaries are unsorted data structures. Write a one-line Python statement returning a list of (*key*, *value*) pairs sorted by key. Assume that all keys have the same data type (why is this important?). Repeat the exercise to produce a list ordered by dictionary *values*.

Q4.3.5 In the television series *The Wire*, drug dealers encrypt telephone numbers with a simple substitution cypher based on the standard layout of the phone keypad. Each digit of the number, with the exception of 5 and 0, is replaced with the corresponding digit on the other side of the 5 key ("jump the five"); 5 and 0 are exchanged. Thus, 555-867-5309 becomes 000-243-0751. Devise a one-line statement to encrypt and decrypt numbers encoded in this way.

Problems

P4.3.1 Use a list comprehension to calculate the *trace* of the matrix `M` (that is, the sum of its diagonal elements). *Hint:* the `sum` built-in function takes an iterable object and sums its values.

P4.3.2 The ROT13 substitution cipher encodes a string by replacing each letter with the letter 13 letters after it in the alphabet (cycling around if necessary). For example, `a` → `n` and `p` → `c`.

a. Given a word expressed as a string of lowercase characters only, use a list comprehension to construct the ROT13-encoded version of that string. *Hint:* Python has a built-in function, `ord`, which converts a character to its Unicode code point (e.g., `ord('a')` returns `97`); another built-in, `chr` is the inverse of `ord` (e.g., `chr(122)` returns `'z'`).

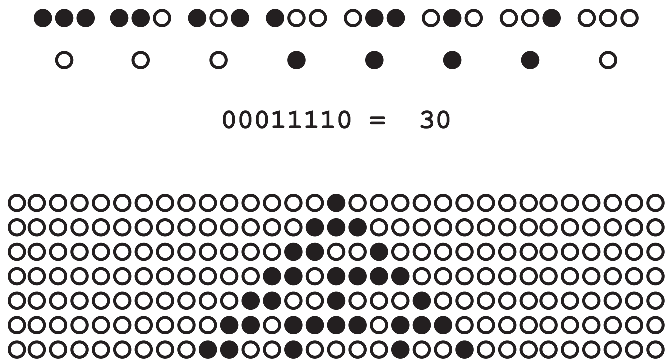


Figure 4.1 Rule 30 of Wolfram's one-dimensional two-state cellular automata and the first seven generations.

b. Extend your list comprehension to encode sentences of words (in lowercase) separated by spaces into a ROT13 sentence (in which the encoded words are also separated by spaces).

P4.3.3 In *A New Kind of Science*,[8] Stephen Wolfram describes a set of simple one-dimensional cellular automata in which each cell can take one of two values: 'on' or 'off'. A row of cells is initialized in some state (e.g., with a single 'on' cell somewhere in the row) and it evolves into a new state according to a rule that determines the subsequent state of a cell ('on' or 'off') from its value and that of its two nearest neighbors. There are $2^3 = 8$ different states for these three "parent" cells taken together and so $2^8 = 256$ different automata rules; that is, the state of cell i in the next generation is determined by the states of cells $i - 1$, i and $i + 1$ in the present generation.

These rules are numbered 0–255 according to the binary number indicated by the eight different outcomes each one specifies for the eight possible parent states. For example, rule 30 produces the outcome (off, off, off, on, on, on, on, off) (or 00011110) from the parent states given in the order shown in Figure 4.1. The evolution of the cells can be illustrated by printing the row corresponding to each generation under its parent as shown in this figure.

Write a program to display the first few rows generated by rule 30 on the command line, starting from a single 'on' cell in the center of a row 80 cells wide. Use an asterisk to indicate an 'on' cell and a space to represent an 'off' cell.

P4.3.4 The file `iban_lengths.txt`, available at scipython.com/ex/add contains two columns of data: a two-letter country code and the length of that country's International Bank Account Number (IBAN):

```
AL 28
AD 24
...
GB 22
```

[8] S. Wofram (2002). *A New Kind of Science*, Wolfram Media.

The code snippet below parses the file into a dictionary of lengths, keyed by the country code:

```
iban_lengths = {}
with open('iban_lengths.txt') as fi:
    for line in fi.readlines():
        fields = line.split()
        iban_lengths[fields[0]] = int(fields[1])
```

Use a `lambda` function and list comprehension to achieve the same goal in (a) two lines, (b) one line.

P4.3.5 The *power set* of a set S, $P(S)$, is the set of all subsets of S, including the empty set and S itself. For example,

$$P(\{1,2,3\}) = \{\{\}, \{1\}, \{2\}, \{3\}, \{1,2\}, \{1,3\}, \{2,3\}, \{1,2,3\}\}.$$

Write a Python program that uses a generator to return the power set of a given set.

Hint: convert your set into an ordered sequence such as a tuple. For each item in this sequence return the power set formed from all subsequent items, inclusive and exclusive of the chosen item. Don't forget to convert the tuples back to sets after you're done.

P4.3.6 The *Brown Corpus* is a collection of 500 samples of (American) English-language text that was compiled in the 1960s for use in the field of computational linguistics. It can be dowloaded from http://nltk.github.com/nltk_data/packages/corpora/brown.zip.

Each sample in the corpus consists of words that have been tagged with their part-of-speech after a forward slash. For example,

```
        The/at football/nn opponent/nn on/in homecoming/nn is/bez ,/, of/in
course/nn ,/, selected/vbn with/in the/at view/nn that/cs
```

Here, `The` has been tagged as an article (`/at`), `football` as a noun (`/nn`) and so on. A full list of the tags is available from the accompanying manual.[9]

Write a program that analyzes the Brown corpus and returns a list of the eight-letter words which feature each possible two-letter combinations *exactly twice*. For example, the two-letter combination *pc* is present in only the words *topcoats* and *upcoming*; *mt* is present only in the words *boomtown* and *undreamt*.

4.4 Operating system services

4.4.1 The `sys` module

The `sys` module provides certain system-specific parameters and functions. Many of them are of interest only to fairly advanced users of less-common Python implementations (the details of how floating point arithmetic is implemented can vary between

[9] This manual is available at www.hit.uib.no/icame/brown/bcm.html though the tags themselves are presented better on the Wikipedia article at http://en.wikipedia.org/wiki/Brown_Corpus.

different systems, for example, but is likely to be the same on all common platforms – see Section 9.1). However, it also provides some that are useful and important: these are described here.

sys.argv

sys.argv holds the command line arguments passed to a Python program when it is executed. *It is a list of strings*. The first item, sys.argv[0], is the name of the program itself. This allows for a degree of interactivity without having to read from configuration files or require direct user input, and means that other programs or shell scripts can call a Python program and pass it particular input values or settings. For example, a simple script to square a given number might be written:

```
# square.py
import sys

n = int(sys.argv[1])
print(n, 'squared is', n**2)
```

(Note that it is necessary to convert the input value into an int, because it is stored in sys.argv as a string.) Running this program from the command line with

```
python square.py 3
```

produces the output

```
3 squared is 9
```

as expected. But because we did not hard-code the value of n, the same program can be run with

```
python square.py 4
```

to produce the output

```
4 squared is 16
```

sys.exit

Calling sys.exit will cause a program to terminate and exit from Python. This happens "cleanly," so that any commands specified in a try statement's finally clause are executed first and any open files are closed. The optional argument to sys.exit can be any object; if it is an integer, it is passed to the shell which, it is assumed, knows what to do with it.[10] For example, 0 usually denotes "successful" termination of the program and nonzero values indicate some kind of error. Passing no argument or None is equivalent to 0. If any other object is specified as an argument to sys.exit, it is passed to stderr, Python's implementation of the standard error stream. A string, for example, appears as an *error message* on the console (unless redirected elsewhere by the shell).

[10] At least if it is in the range 0–127; undefined results could be produced for values outside this range.

Example E4.13 A common way to help users with scripts that take command line arguments is to issue a usage message if they get it wrong, as in the following code example.

Listing 4.3 Issuing a usage message for a script taking command line arguments

```
# square.py
import sys

try:
    n = int(sys.argv[1])
except (IndexError, ValueError):
    sys.exit('Please enter an integer, <n>, on the command line.\nUsage: '
             'python {:s} <n>'.format(sys.argv[0]))
print(n, 'squared is', n**2)
```

The error message here is reported and the program exits if no command line argument was specified (and hence indexing `sys.argv[1]` raises an `IndexError`) or the command line argument string does not evaluate to an integer (in which case the `int` cast will raise a `ValueError`).

```
$ python eg4-usage.py hello
Please enter an integer, <n>, on the command line.
Usage: python eg4-usage.py <n>

$ python eg4-usage.py 5
5 squared is 25
```

4.4.2 The `os` module

The `os` module provides various operating system interfaces in a platform-independent way. Its many functions and parameters are described in full in the official documentation,[11] but some of the more important are described in this section.

Process information

The Python *process* is the particular instance of the Python application that is executing your program (or providing a Python shell for interactive use). The `os` module provides a number of functions for retrieving information about the context in which the Python process is running. For example, `os.uname()` returns information about the operating system running Python and the network name of the machine running the process.

One function is of particular use: `os.getenv(key)` returns the value of the environment variable *`key`* if it exists (or `None` of it doesn't). Many environment variables are system specific, but commonly include

[11] http://docs.python.org/3/library/os.html.

- `HOME`: the path to the user's home directory,
- `PWD`: the current working directory,
- `USER`: the current user's username and
- `PATH`: the system path environment variable.

For example, on my system:

```
>>> os.getenv('HOME')
'/Users/christian'
```

File system commands

It is often useful to be able to navigate the system directory tree and manipulate files and directories from within a Python program. The `os` module provides the functions listed in Table 4.4 to do just this. There are, of course, inherent dangers: your Python program can do anything that your user can, including renaming and deleting files.

Pathname manipulations

The `os.path` module provides a number of useful functions for manipulating pathnames. The version of this library installed with Python will be the one appropriate for the operating system that it runs on (e.g., on a Windows machine, path name components are separated by the backslash character, '`\`', whereas on Unix and Linux systems, the (forward) slash character, '`/`' is used.

Common usage of the `os.path` module's functions are to find the filename from a path (`basename`), test to see if a file or directory exists (`exists`), join strings together to make a path (`join`), split a filename into a 'root' and an 'extension' (`splitext`) and to find the time of last modification to a file (`getmtime`). Such common applications are described briefly in Table 4.5.

Table 4.4 `os` module: file system commands

Function	Description
`os.listdir(`*`path='.'`*`)`	List the entries in the directory given by *`path`* (or the current working directory if this is not specified).
`os.remove(`*`path`*`)`	Delete the file *`path`* (raises an `OSError` if *`path`* is a directory; use `os.rmdir` instead).
`os.rename(`*`old_name`*`, `*`new_name`*`)`	Rename the file or directory *`old_name`* to *`new_name`*. If a file with the name *`new_name`* already exists, *it will be overwritten* (subject to user-permissions).
`os.rmdir(`*`path`*`)`	Delete the directory *`path`*. If the directory is not empty, an `OSError` is raised.
`os.mkdir(`*`path`*`)`	Create the directory named *`path`*.
`os.system(`*`command`*`)`	Execute *`command`* in a subshell. If the command generates any output, it is redirected to the interpreter standard output stream, `stdout`.

Table 4.5 `os.path` module: common pathname manipulations

Function	Description
`os.path.basename(path)`	Return the basename of the pathname *path* giving a relative or absolute path to the file: this usually means the filename.
`os.path.dirname(path)`	Return the directory of the pathname *path*.
`os.path.exists(path)`	Return `True` if the directory or file *path* exists, and `False` otherwise.
`os.path.getmtime(path)`	Return the time of last modification of *path*.
`os.path.getsize(path)`	Return the size of *path* in bytes.
`os.path.join(path1, path2, ...)`	Return a pathname formed by joining the path components *path1*, *path2*, etc. with the directory separator appropriate to the operating system being used.
`os.path.split(path)`	Split *path* into a directory and a filename, returned as a tuple (equivalent to calling `dirname` and `basename`) respectively.
`os.path.splitext(path)`	Split *path* into a 'root' and an 'extension' (returned as a tuple pair).

Some examples referring to a file `/home/brian/test.py`:

```
>>> os.path.basename('/home/brian/test.py')
'test.py'                  # Just the filename

>>> os.path.dirname('/home/brian/test.py')
'/home/brian'              # Just the directory

>>> os.path.split('/home/brian/test.py')
('/home/brian', 'test.py')  # Directory and filename in a tuple

>>> os.path.splitext('/home/brian/test.py')
('/home/brian/test', '.py') # File path stem and extension in a tuple

>>> os.path.join(os.getenv('HOME'), 'test.py')
'/home/brian/test.py'       # Join directories and/or filename

>>> os.path.exists('/home/brian/test.py')
False                       # File does not exist!
```

Trying to call some of these functions on a path that does not exist will cause a `FileNotFoundError` exception to be raised (which could be caught within a `try ... except` clause, of course).

Example E4.14 Suppose you have a directory of data files identified by filenames containing a date in the form `data-DD-Mon-YY.txt` where `DD` is the two-digit day number, `Mon` is the three-letter month abbreviation and `YY` is the last two digits of the year, for example `'02-Feb-10'`. The following program converts the filenames into the form `data-YYYY-MM-DD.txt` so that an alphanumeric ordering of the filenames puts them in chronological order.

Listing 4.4 Renaming data files by date

```
# eg4-osmodule.py
import os
import sys

months = ['jan', 'feb', 'mar', 'apr', 'may', 'jun',
          'jul', 'aug', 'sep', 'oct', 'nov', 'dec']

dir_name = sys.argv[1]
for filename in os.listdir(dir_name):
    # filename is expected to be in the form 'data-DD-MMM-YY.txt'
    d, month, y = int(filename[5:7]), filename[8:11], int(filename[12:14])
    m = months.index(month.lower())+1       ❶

    newname = 'data-20{:02d}-{:02d}-{:02d}.txt'.format(y, m, d)
    newpath = os.path.join(dir_name, newname)
    oldpath = os.path.join(dir_name, filename)
    print(oldpath, '->', newpath)
    os.rename(oldpath, newpath)
```

❶ We get the month number from the index of corresponding abbreviated month name in the list `months`, adding 1 because Python list indexes start at 0.

For example, given a directory `testdir` containing the following files:

```
data-02-Feb-10.txt
data-10-Oct-14.txt
data-22-Jun-04.txt
data-31-Dec-06.txt
```

the command `python eg4-osmodule.py testdir` produces the output

```
testdir/data-02-Feb-10.txt -> testdir/data-2010-02-02.txt
testdir/data-10-Oct-14.txt -> testdir/data-2014-10-10.txt
testdir/data-22-Jun-04.txt -> testdir/data-2004-06-22.txt
testdir/data-31-Dec-06.txt -> testdir/data-2006-12-31.txt
```

See also Problem 4.4.4 and the `datetime` module (Section 4.5.3).

4.4.3 Exercises

Problems

P4.4.1 Modify the hailstone sequence generator of Exercise P2.5.7 to generate the hailstone sequence starting at any positive integer that the user provides on the command line (use `sys.argv`). Handle the case where the user forgets to provide n or provides an invalid value for n gracefully.

P4.4.2 The *Haversine* formula gives the shortest (great-circle) distance, d, between two points on a sphere of radius R from their longitudes (λ_1, λ_2) and latitudes (ϕ_1, ϕ_2):

$$d = 2r \arcsin\left(\sqrt{\mathrm{haversin}(\phi_2 - \phi_1) + \cos\phi_1 \cos\phi_2 \mathrm{haversin}(\lambda_2 - \lambda_1)}\right),$$

where the *haversine* function of an angle is defined by

$$\text{haversin}(\alpha) = \sin^2\left(\frac{\alpha}{2}\right).$$

Write a program to calculate the shortest distance in km between two points on the surface of the Earth (considered as a a sphere of radius 6378.1 km) given as two command line arguments, each of which is a comma-separated pair of latitude, longitude values in degrees. For example, the distance between Paris and Rome is given by executing:

```
python greatcircle.py 48.9,2.4 41.9,12.5
1107 km
```

P4.4.3 Write a Python program to create a directory, `test`, in the user's home directory and to populate it with 20 Scalable Vector Graphics (SVG) files depicting a small, filled red circle inside a large, black, unfilled circle. For example,

```
<?xml version="1.0" encoding="utf-8"?>
                 <svg xmlns="http://www.w3.org/2000/svg"
                      xmlns:xlink="http://www.w3.org/1999/xlink"
                      width="500" height="500" style="background: #ffffff">
<circle cx="250.0" cy="250.0" r="200" style="stroke: black; stroke-width: 2px;
                                                               fill: none;"/>
<circle cx="430.0" cy="250.0" r="20" style="stroke: red; fill: red;"/>
</svg>
```

Each file should move the red circle around the inside rim of the larger circle so that the 20 files together could form an animation.

One way to achieve this is to use the free ImageMagick software (www.imagemagick.org/). Ensure the SVG files are named `fig00.svg`, `fig01.svg`, etc. and issue the following command from your operating system's command line:

```
convert -delay 5 -loop 0 fig*.svg animation.gif
```

to produce an animated GIF image.

P4.4.4 Modify the program of Example E4.14 to catch the following errors and handle them gracefully:

- User does not provide a directory name on the command line (issue a usage message);
- The directory does not exist;
- The name of a file in the directory does not have the correct format;
- The filename is in the correct format but the month abbreviation is not recognized.

Your program should terminate in the first two cases and skip the file in the second two.

4.5 Modules and packages

As we have seen, Python is quite a modular language and has functionality beyond the core programming essentials (the built-in methods and data structures we have

encountered so far) that is made available to a program through the `import` statement. This statement makes reference to *modules* that are ordinary Python files containing definitions and statements. Upon encountering the line

```
import <module>
```

the Python interpreter executes the statements in the file `<module>.py` and enters the module name `<module>` into the current namespace, so that the attributes it defines are available with the "dotted syntax": `<module>.<attribute>`.

Defining your own module is as simple as placing code within a file `<module>.py`, which is somewhere the Python interpreter can find it (for small projects, usually just the same directory as the program doing the importing). Note that because of the syntax of the `import` statement, you should avoid naming your module anything that isn't a valid Python identifier (see Section 2.2.3). For example, the filename `<module>.py` should not contain a hyphen or start with a digit. Do not give your module the same name as any built-in modules (such as `math` or `random`) because these get priority when Python imports.

A Python *package* is simply a structured arrangement of modules within a directory on the file system. Packages are the natural way to organize and distribute larger Python projects. To make a package, the module files are placed in a directory, along with a file named `__init__.py`. This file is run when the package is imported and may perform some initialization and its own imports. It may be an empty file (zero bytes long) if no special initialization is required, but it must exist for the directory to be considered by Python to be a package.

For example, the NumPy package (see Chapter 6) exists as the following directory (some files and directories have been omitted for clarity):

```
numpy/
    __init__.py
    core/
    fft/
        __init__.py
        fftpack.py
        info.py
        ...
    linalg/
        __init__.py
        linalg.py
        info.py
        ...
    polynomial/
        __init__.py
        chebyshev.py
        hermite.py
        legendre.py
        ...
    random/
    version.py
    ...
```

Table 4.6 Python modules and packages

Module / Package	Description
os, sys	Operating system services, as described in Section 4.4
math, cmath	Mathematical functions, as introduced in Section 2.2.2
random	Random number generator (see Section 4.5.1)
collections	Data types for containers that extend the functionality of dictionaries, tuples, etc.
itertools	Tools for efficient iterators that extend the functionality of simple Python loops
glob	Unix-style pathname pattern expansion
datetime	Parsing and manipulating dates and times (see Section 4.5.3)
fractions	Rational number arithmetic
re	Regular expressions
argparse	Parser for command line options and arguments
urllib	URL (including web pages) opening, reading and parsing (see Section 4.5.2)
* Django (django)	A popular web application framework
* pyparsing	Lexical parser for simple grammars
pdb	The Python debugger
logging	Python's built-in logging module
xml, lxml	XML parsers
* VPython (visual)	Three-dimensional visualization
unittest	Unit testing framework for systematically testing and validating individual units of code (see Section 9.3.4)
* NumPy (numpy)	Numerical and scientific computing (described in detail in Chapter 6)
* SciPy (scipy)	Scientific computing algorithms (described in detail in Chapter 8)
* matplotlib, pylab	Plotting (see Chapters 3 and 7)
* SymPy (sympy)	Symbolic computation (computer algebra)
* pandas	Data manipulation and analysis with table-like data structures
* scikit-learn	Machine learning
* Beautiful Soup (beautifulsoup)	HTML parser, with handling of malformed documents

Thus, for example, polynomial is a subpackage of the numpy package containing several modules, including legendre, which may be imported as

```
import numpy.polynomial.legendre
```

To avoid having to use this full dotted syntax in actually referring to its attributes, it is convenient to use

```
from numpy.polynomial import legendre
```

Table 4.6 lists some of the major, freely available Python modules and packages for general programming applications as well as for numerical and scientific work. Some are installed with the core Python distribution (the *Standard Library*);[12] where

[12] A complete list of the components of the Standard Library is at https://docs.python.org/3/library/index.html.

indicated; others can be downloaded and installed separately. Before implementing your own algorithm, check that it isn't included in an existing Python package.

4.5.1 The `random` module

For simulations, modeling and some numerical algorithms it is often necessary to generate random numbers from some distribution. The topic of random-number generation is a complex and interesting one, but the important aspect for our purposes is that, in common with most other languages, Python implements a *pseudorandom number generator* (PRNG). This is an algorithm that generates a sequence of numbers that approximates the properties of "truly" random numbers. Such sequences are determined by an originating *seed* state and are always the same following the same seed: in this sense they are deterministic. This can be a good thing (so that a calculation involving random numbers can be reproduced) or a bad thing (e.g., if used for cryptography, where the random sequence must be kept secret). Any PRNG will yield a sequence that eventually repeats, and a good generator will have a long period. The PRNG implemented by Python is the *Mersenne Twister*, a well-respected and much-studied algorithm with a period of $2^{19937} - 1$ (a number with more than 6,000 digits in base 10).

Generating random numbers

The random number generator can be seeded with any *hashable* object (e.g., an *immutable* object such as an integer). When the module is first imported, it is seeded with a representation of the current system time (unless the operating system provides a better source of a random seed). The PRNG can be reseeded at any time with a call to `random.seed`.

The basic random number method is `random.random`. It generates a random number selected from the uniform distribution in the semi-open interval [0, 1) – that is, including 0 but not including 1.

```
>>> import random
>>> random.random()      # PRNG seeded 'randomly'
0.5204514767709216
>>> random.seed(42)      # Seed the PRNG with a fixed value
>>> random.random()
0.6394267984578837
>>> random.random()
0.025010755222666936
...
>>> random.seed(42)      # Reseed with the same value as before ...
>>> random.random()
0.6394267984578837       # ... and the sequence repeats.
>>> random.random()
0.025010755222666936
```

Calling `random.seed()` with no argument reseeds the PRNG with a 'random' value as when the `random` module is first imported.

To select a random floating point number, N, from a given range, $a \le N \le b$, use `random.uniform(a, b)`:

```
>>> random.uniform(-2., 2.)
-0.899882726523523
>>> random.uniform(-2., 2.)
-1.107157047404709
```

The `random` module has several methods for drawing random numbers from nonuniform distributions – see the documentation[13] – here we mention the most important of them.

To return a number from the normal distribution with mean `mu` and standard deviation `sigma`, use `random.normalvariate(mu, sigma)`:

```
>>> random.normalvariate(100, 15)
118.82178896586194
>>> random.normalvariate(100, 15)
97.92911405885782
```

To select a random *integer*, N, in a given range, $a \le N \le b$, use `random.randint (a, b)` method:

```
>>> random.randint(5, 10)
7
>>> random.randint(5, 10)
10
```

Random sequences

Sometimes you may wish to select an item at random from a sequence such as a `list`. This is what the method `random.choice` does:

```
>>> seq = [10, 5, 2, 'ni', -3.4]
>>> random.choice(seq)
-3.4
>>> random.choice(seq)
'ni'
```

Another method, `random.shuffle`, randomly shuffles (permutes) the items of the sequence *in place*:

```
>>> random.shuffle(seq)
>>> seq
[10, -3.4, 2, 'ni', 5]
```

Note that because the random permutation is made in place, the sequence must be mutable: you can't, for example, shuffle `tuples`.

Finally, to draw a `list` of *k* unique elements from a sequence or set (without replacement) `population`, there is `random.sample(population, k)`:

```
>>> raffle_numbers = range(1, 100001)
>>> winners = random.sample(raffle_numbers, 5)
>>> winners
[89734, 42505, 7332, 30022, 4208]
```

[13] https://docs.python.org/3/library/random.html.

The resulting list is in selection order (the first-indexed element is the first drawn) so that one could, for example, without bias declare ticket number 89734 to be the jackpot winner and the remaining four tickets second-placed winners.

Example E4.15 The *Monty Hall problem* is a famous conundrum in probability which takes the form of a hypothetical game show. The contestant is presented with three doors; behind one is a car and behind each of the other two is a goat. The contestant picks a door and then the game show host opens a different door to reveal a goat. The host knows which door conceals the car. The contestant is then invited to switch to the other closed door or stick with his or her initial choice.

Counterintuitively, the best strategy for winning the car is to switch, as demonstrated by the following simulation.

Listing 4.5 The Monty Hall problem

```
# eg4-montyhall.py
import random

def run_trial(switch_doors, ndoors=3):
    """
    Run a single trial of the Monty Hall problem, with or without switching
    after the game show host reveals a goat behind one of the unchosen doors.
    (switch_doors is True or False). The car is behind door number 1 and the
    game show host knows that. Returns True for a win, otherwise returns False.

    """

    # Pick a random door out of the ndoors available
    chosen_door = random.randint(1, ndoors)
    if switch_doors:
        # Reveal a goat
        revealed_door = 3 if chosen_door==2 else 2
        # Make the switch by choosing any other door than the initially
        # selected one and the one just opened to reveal a goat.
        available_doors = [dnum for dnum in range(1,ndoors+1)
                                if dnum not in (chosen_door, revealed_door)]
        chosen_door = random.choice(available_doors)

    # You win if you picked door number 1
❶   return chosen_door == 1

def run_trials(ntrials, switch_doors, ndoors=3):
    """
    Run ntrials iterations of the Monty Hall problem with ndoors doors, with
    and without switching (switch_doors = True or False). Returns the number
    of trials which resulted in winning the car by picking door number 1.

    """

    nwins = 0
    for i in range(ntrials):
        if run_trial(switch_doors, ndoors):
            nwins += 1
    return nwins
```

```
ndoors, ntrials = 3, 10000
nwins_without_switch = run_trials(ntrials, False, ndoors)
nwins_with_switch = run_trials(ntrials, True, ndoors)

print('Monty Hall Problem with {} doors'.format(ndoors))
print('Proportion of wins without switching: {:.4f}'
            .format(nwins_without_switch/ntrials))
print('Proportion of wins with switching: {:.4f}'
            .format(nwins_with_switch/ntrials))
```

❶ Without loss of generality, we can place the car behind door number 1, leaving the contestant initially to choose any door at random.

To make the code a little more interesting, we have allowed for a variable number of doors in the simulation (but only one car).

```
Monty Hall Problem with 3 doors
Proportion of wins without switching: 0.3334
Proportion of wins with switching: 0.6737
```

4.5.2 ◇ The `urllib` package

The `urllib` package in Python 3 is a set of modules for opening and retrieving the content referred to by uniform resource locators (URLs), typically web addresses accessed with HTTP(S) or FTP. Here we give a very brief introduction to its use.

Opening and reading URLs

To obtain the content at a URL using HTTP you first need to make an HTTP *request* by creating a `Request` object. For example,

```
import urllib.request
req = urllib.request.Request('http://www.wikipedia.org')
```

The `Request` object allows you to pass data (using GET or POST) and other information about the request (metadata passed through the HTTP headers – see later). For a simple request, however, one can simply open the URL immediately as a file-like object with `urlopen()`:

```
response = urllib.request.urlopen(req)
```

It's a good idea to catch the two main types of exception that can arise from this statement. The first type, `URLError`, results if the server doesn't exist or if there is no network connection; the second type, `HTTPError`, occurs when the server returns an error code (such as *404: Page Not Found*). These exceptions are defined in the `urllib.error` module.

```
from urllib.error import URLError, HTTPError
try:
    response = urllib.request.urlopen(req)
except HTTPError as e:
    print('The server returned error code', e.code)
except URLError as e:
```

```
    print('Failed to reach server at {} for the following reason:\n{}'
            .format(url, e.reason))
else:
    # the response came back OK
```

Assuming the `urlopen()` worked, there is often nothing more to do than simply read the content from the response:

```
content = response.read()
```

The content will be returned as a *bytestring*. To decode it into a Python (Unicode) string you need to know how it is encoded. A good resource will include the character set used in the `Content-Type` HTTP header. This can be used as follows:

```
charset = response.headers.get_content_charset()
html = content.decode(charset)
```

where `html` is now a decoded Python Unicode string. If no character set is specified in the headers returned, you may have to guess (e.g., set `charset='utf-8'`).

GET and POST requests

It is often necessary to pass data along with the URL to retrieve content from a server. For example, when submitting an HTML form from a web page, the values corresponding to the entries you have made are encoded and passed to the server according to either the GET or POST protocols.

The `urllib.parse` module allows you to encode data from a Python dictionary into a form suitable for submission to a web server. To take an example from the Wikipedia API using a GET request:

```
>>> url = 'http://wikipedia.org/w/api.php'
>>> data = {'page': 'Monty_Python', 'prop': 'text', 'action': 'parse', 'section': 0}
>>> encoded_data = urllib.parse.urlencode(data)
>>> full_url = url + '?' + encoded_data
>>> full_url
'http://wikipedia.org/w/api.php?page=Monty_Python&prop=text&action=parse
         &section=0'
>>> req = urllib.request.Request(full_url)
>>> response = urllib.request.urlopen(req)
>>> html = response.read().decode('utf-8')
```

To make a POST request, instead of appending the encoded data to the string `<url>?`, pass it to the `Request` constructor directly:

```
req = urllib.request.Request(url, encoded_data)
```

4.5.3 The `datetime` module

Python's `datetime` module provides classes for manipulating dates and times. There are many subtle issues surrounding the handling of such data (time zones,

different calendars, Daylight Saving Time etc.,) and full documentation is available online;[14] here we provide an overview of only the most common uses.

Dates

A `datetime.date` object represents a particular day, month and year in an idealized calendar (the current Gregorian calendar is assumed to be in existence for all dates, past and future). To create a `date` object, pass valid year, month and day numbers explicitly, or call the `date.today` constructor:

```
>>> from datetime import date
>>> birthday = date(2004, 11, 5)     # OK

>>> notadate = date(2005, 2, 29)     # Oops: 2005 wasn't a leap year


Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: day is out of range for month

>>> today = date.today()
>>> today
datetime.date(2014, 12, 6)   # (for example)
```

Dates between 1/1/1 and 31/12/9999 are accepted. Parsing dates to and from strings is also supported (see `strptime` and `strftime`).

Some more useful `date` object methods:

```
>>> birthday.isoformat()     # ISO 8601 format: YYYY-MM-DD
'2004-11-05'

>>> birthday.weekday()       # Monday = 0, Tuesday = 1, ..., Sunday = 6
4   # (Friday)

>>> birthday.isoweekday()    # Monday = 1, Tuesday = 2, ..., Sunday = 7
5

>>> birthday.ctime()         # C-standard time output
'Fri Nov  5 00:00:00 2004'
```

`dates` can also can be compared (chronologically):

```
>>> birthday < today
True

>>> today == birthday
False
```

Times

A `datetime.time` object represents a (local) time of day to the nearest microsecond. To create a `time` object, pass the number of hours, minutes, seconds and microseconds (in that order; missing values default to zero).

[14] https://docs.python.org/3/library/datetime.html.

```
>>> from datetime import time
>>> lunchtime = time(hour=13, minute=30)
>>> lunchtime
datetime.time(13, 30)

>>> lunchtime.isoformat()       # ISO 8601 format: HH:MM:SS if no microseconds
'13:30:00'

>>> precise_time = time(4,46,36,501982)
>>> precise_time.isoformat()    # ISO 8601 format: HH:MM:SS.mmmmmm
'04:46:36.501982'

>>> witching_hour = time(24)    # Oops: hour must satisfy 0 <= hour < 24

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: hour must be in 0..23
```

datetime objects

A datetime.datetime object contains the information from both the date and time objects: year, month, day, hour, minute, second, microsecond. As well as passing values for these quantities directly to the datetime constructor, the methods today (returning the current date) and now (returning the current date and time) are available:

```
>>> from datetime import datetime   # (a notoriously ugly import)
>>> now = datetime.now()
>>> now
datetime.datetime(2014, 12, 6, 12, 4, 51, 763063)

>>> now.isoformat()
'2014-12-06T12:04:51.763063'

>>> now.ctime()
'Sat Dec  6 12:04:51 2014'
```

Date and time formatting

date, time and datetime objects support a method, strftime to output their values as a string formatted according to a syntax set using the format specifiers listed in Table 4.7.

```
>>> birthday.strftime('%A, %d %B %Y')
'Friday, 05 November 2004'

>>> now.strftime('%I:%M:%S on %d/%m/%y')
'12:04:51 on 06/12/14'
```

The reverse process, parsing a string into a datetime object is the purpose of the strptime method:

```
>>> launch_time = datetime.strptime('09:32:00 July 16, 1969',
                                    '%H:%M:%S %B %d, %Y')
>>> print(launch_time)
1969-07-16 09:32:00
```

Table 4.7 `strftime` and `strptime` format specifiers. Note that many of these are locale-dependent (e.g., on a German-language system, `%A` will yield `Sonntag`, `Montag`, etc.).

Specifier	Description
`%a`	Abbreviated weekday (`Sun`, `Mon`, etc.)
`%A`	Full weekday (`Sunday`, `Monday`, etc.)
`%w`	Weekday number (0=Sunday, 1=Monday, ..., 6=Saturday).
`%d`	Zero-padded day of month: 01, 02, 03, ..., 31.
`%b`	Abbreviated month name (`Jan`, `Feb`, etc.)
`%B`	Full month name (`January`, `February`, etc.)
`%m`	Zero-padded month number: 01, 02, ..., 12.
`%y`	Year without century (two-digit, zero-padded): 01, 02, ..., 99.
`%Y`	Year with century (four-digit, zero-padded): 0001, 0002, ... 9999.
`%H`	24-hour clock hour, zero-padded: 00, 01, ..., 23.
`%I`	12-hour clock hour, zero-padded: 00, 01, ..., 12.
`%p`	AM or PM (or locale equivalent).
`%M`	Minutes (two-digit, zero-padded): 00, 01, ..., 59.
`%S`	Seconds (two-digit, zero-padded): 00, 01, ..., 59.
`%f`	Microseconds (six-digit, zero-padded): 000000, 000001, ..., 999999.
`%%`	The literal `%` sign.

```
>>> print(launch_time.strftime('%I:%M %p on %A, %d %b %Y'))
09:32 AM on Wednesday, 16 Jul 1969
```

4.6 ◊ An introduction to object-oriented programming

4.6.1 Object-oriented programming basics

Structured programming styles may be broadly divided into two categories: *procedural* and *object-oriented*. The programs we have looked at so far in this book have been *procedural* in nature: we have written functions (of the sort that would be called procedures or subroutines in other languages) that are called, passed data, and which return values from their calculations. The functions we have defined do not hold their own data or remember their state in between being called, and we haven't modified them after defining them.

An alternative programming paradigm that has gained popularity through the use of languages such as C++ and Java is *object-oriented programming*. In this context, an *object* represents a concept of some sort which holds data about itself (*attributes*) and defines functions (*methods*) for manipulating data. That manipulation may cause a change in the object's state (i.e., it may change some of the object's attributes). An object is created (*instantiated*) from a "blueprint" called a *class*, which dictates its behavior by defining its attributes and methods.

In fact, as we have already pointed out, everything in Python is an object. So, for example, a Python string is an instance of the `str` class. A `str` object possesses its

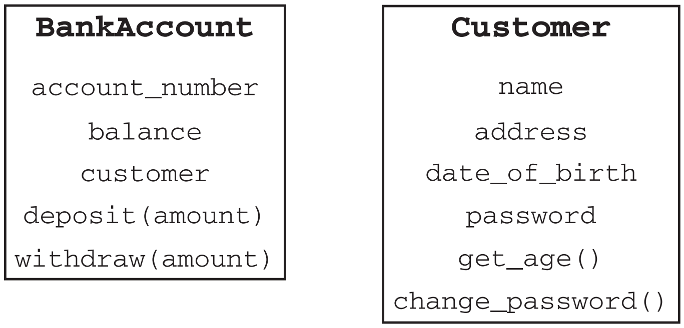


Figure 4.2 Basic classes representing a bank account and a customer.

own data (the sequence of characters making up the string) and provides (*"exposes"*) a number of methods for manipulating that data. For example, the `capitalize` method returns a new string object created from the original string by capitalizing its first letter; the `split` method returns a list of strings by splitting up the original string:

```
>>> a = 'hello, aloha, goodbye, aloha'
>>> a.capitalize()
'Hello, aloha, goodbye, aloha'
>>> a.split(',')
['hello', ' aloha', ' goodbye', ' aloha']
```

Even indexing a string is really to call the method `__getitem__`:

```
>>> b = [10, 20, 30, 40, 50]
>>> b.__getitem__(4)
50
```

That is, `a[4]` is equivalent to `a.__getitem__(4)`.[15]

Part of the popularity of object-oriented programming, at least for larger projects, stems from the way it helps conceptualize the problem that a program aims to solve. It is often possible to break a problem down into units of data and operations that it is appropriate to carry out on that data. For example, a retail bank deals with people who have bank accounts. A natural object-oriented approach to managing a bank would be to define a `BankAccount` class, with attributes such as an account number, balance and owner, and a second, `Customer` class with attributes such as a name, address, and date of birth. The `BankAccount` class might have methods for allowing (or forbidding) transactions depending on its balance and the `Customer` class might have methods for calculating the customer's age from their date of birth for example (see Figure 4.2).

An important aspect of object-oriented programming is *inheritance*. There is often a relationship between objects which takes the form of a hierarchy. Typically, a general type of object is defined by a base class, and then customized classes with more specialized functionality are derived from it. In our bank example, there may be different kinds of bank accounts: savings accounts, current (checking) accounts, etc. Each is derived from a generic base bank account, which might simply define basic attributes such as a balance and an account number. The more specialized bank account classes *inherit*

[15] The double-underscore syntax usually denotes a name with some special meaning to Python.

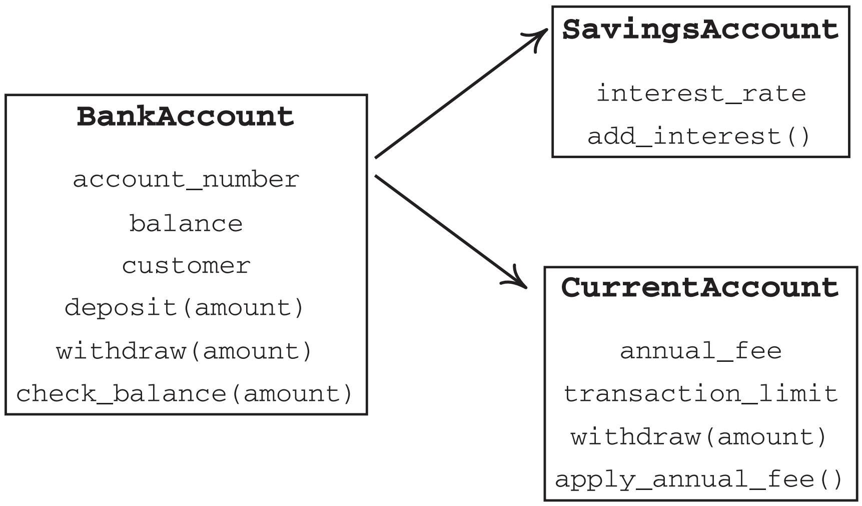


Figure 4.3 Two classes derived from an abstract base class: `SavingsAccount` and `CurrentAccount` *inherit* methods and attributes from `BankAccount` but also customize and extend its functionality.

the properties of the base class but may also customize them by overriding (redefining) one or more methods and may also add their own attributes and methods. This helps structure the program and encourages *code reuse* – there is no need to declare an account number separately for both savings and current accounts because both classes inherit one automatically from the base class. If a base class is not to be instantiated itself, but serves only as a template for the derived classes, it is called an *abstract class*.

In Figure 4.3, the relationship between the base class and two derived subclasses is depicted. The base class, `BaseAccount`, defines some attributes (`account_number`, `balance` and `customer`) and methods (such as `deposit` and `withdraw`) common to all types of account, and these are inherited by the subclasses. The subclass `SavingsAccount` adds an attribute and a method for handling interest payments on the account; the subclass `CurrentAccount` instead adds two attributes describing the annual account fee and transaction withdrawal limit, and overrides the base `withdraw` method, perhaps to check that the transaction limit has not been reached before a withdrawal is allowed.

4.6.2 Defining and using classes in Python

A class is defined using the `class` keyword and indenting the body of statements (attributes and methods) in a block following this declaration. It is conventional to give classes names written in *CamelCase*. It is a good idea to follow the `class` statement with a docstring describing what it is that the class does (see Section 2.7.1). Class methods are defined using the familiar `def` keyword, but the first argument to each

method should be a variable named `self`[16] – this name is used to refer to the object itself when it wants to call its own methods or refer to attributes, as we shall see.

In our example of a bank account, the base class could be defined as follows:

Listing 4.6 The definition of the abstract base class, `BankAccount`

```
# bank_account.py

class BankAccount:
    """ A abstract base class representing a bank account."""
    currency = '$'

    def __ init __(self, customer, account_number, balance=0):
        """
        Initialize the BankAccount class with a customer, account number
        and opening balance (which defaults to 0.)

        """

        self.customer = customer
        self.account_number = account_number
        self.balance = balance

    def deposit(self, amount):
        """ Deposit amount into the bank account."""
        if amount > 0:
            self.balance += amount
        else:
            print('Invalid deposit amount:', amount)

    def withdraw(self, amount):
        """
        Withdraw amount from the bank account, ensuring there are sufficient
        funds.

        """
        if amount > 0:
            if amount > self.balance:
                print('Insufficient funds')
            else:
                self.balance -= amount
        else:
            print('Invalid withdrawal amount:', amount)
```

To use this simple class, we can save the code defining it as `bank_account.py` and import it into a new program or the interactive Python shell with

```
from bank_account import BankAccount
```

This new program can now create `BankAccount` objects and manipulate them by calling the methods described earlier.

[16] Actually, it could be named anything, but `self` is almost universally used.

Instantiating the object

An *instance* of a class is created with the syntax `object = ClassName(args)`. You may want to require that an object instantiated from a class should initialize itself in some way (perhaps by setting attributes with appropriate values) – such initialization is carried out by the special method `__init__` which receives any arguments, *args*, specified in this statement.

In our example, an account is opened by creating a `BankAccount` object, passing the name of the account owner (customer), an account number and, optionally, an opening balance (which defaults to 0 if not provided):

```
my_account = BankAccount('Joe Bloggs', 21457288)
```

We will replace the string `customer` with a `Customer` object in Example E4.16.

Methods and attributes

The class defines two methods: one for depositing a (positive) amount of money and one for withdrawing money (if the amount to be withdrawn is both positive and not greater than the account balance).

The `BankAccount` class possesses two different kinds of attribute: `self.customer`, `self.account_number` and `self.balance` are *instance variables*: they can take different values for different objects created from the `BankAccount` class. Conversely, the variable `currency` is a *class variable*: this variable is defined inside the class but outside any of its methods and is shared by all instances of the class.

Both attributes and methods are accessed using the `object.attr` notation. For example,

```
>>> my_account.account_number      # access an attribute of my_account
21457288
>>> my_account.deposit(64)         # call a method of my_account
>>> my_account.balance
64
```

Let's add a third method, for printing the balance of the account. This must be defined inside the `class` block:

```
    def check_balance(self):
        """ Print a statement of the account balance. """
        print('The balance of account number {:d} is {:s}{:f.2}'
                .format(self.account_number, self.currency, self.balance))
```

Example E4.16 We now define the `Customer` class described in class diagram of Figure 4.2: an instance of this class will become the `customer` attribute of the `BankAccount` class. Note that it was possible to instantiate a `BankAccount` object by passing a string literal as `customer`. This is a consequence of Python's dynamic typing: no check is automatically made that the object passed as an argument to the class constructor is of any particular type.

The following code defines a `Customer` class and should be saved to a file called `customer.py`:

```
from datetime import datetime
class Customer:
    """ A class representing a bank customer. """

    def __init__(self, name, address, date_of_birth):
        self.name = name
        self.address = address
        self.date_of_birth = datetime.strptime(date_of_birth, '%Y-%m-%d')
        self.password = '1234'

    def get_age(self):
        """ Calculates and returns the customer's age. """
        today = datetime.today()
        try:
            birthday = self.date_of_birth.replace(year=today.year)
        except ValueError:
            # birthday is 29 Feb but today's year is not a leap year
            birthday = self.date_of_birth.replace(year=today.year,
                                            day=self.date_of_birth.day - 1)
        if birthday > today:
            return today.year - self.date_of_birth.year - 1
        return today.year - self.date_of_birth.year
```

Then we can pass `Customer` objects to our `BankAccount` constructor:

```
>>> from bank_account import BankAccount
>>> from customer import Customer
>>>
>>> customer1 = Customer('Helen Smith', '76 The Warren, Blandings, Sussex',
                         '1976-02-29')
>>> account1 = BankAccount(customer1, 21457288, 1000)
>>> account1.customer.get_age()
39
>>> print(account1.customer.address)
76 The Warren, Blandings, Sussex
```

4.6.3 Class inheritance in Python

A subclass may be derived from one or more other base classes with the syntax:

```
class SubClass(BaseClass1, BaseClass2, ...):
```

We will now define the two derived classes (or *subclasses*) illustrated in Figure 4.3 from the base `BankAccount` class. They can be defined in the same file that defines `BankAccount` or in a different Python file which imports `BankAccount`.

```
class SavingsAccount(BankAccount):
    """ A class representing a savings account. """

    def __init__(self, customer, account_number, interest_rate, balance=0):
        """ Initialize the savings account. """
❶       self.interest_rate = interest_rate
```

```
❷           super().__init__(customer, account_number, balance)

    def add_interest(self):
        """ Add interest to the account at the rate self.interest_rate. """

        self.balance *= (1. + self.interest_rate / 100)
```

❶ The `SavingsAccount` class adds a new attribute, `interest_rate`, and a new method, `add_interest` to its base class, and overrides the `__init__` method to allow `interest_rate` to be set when a `SavingsAccount` is instantiated.
❷ Note that the new `__init__` method calls the base class's `__init__` method in order to set the other attributes: the built-in function `super` allows us to refer to the parent base class.[17] Our new `SavingsAccount` might be used as follows:

```
>>> my_savings = SavingsAccount('Matthew Walsh', 41522887, 5.5, 1000)
>>> my_savings.check_balance()
The balance of account number 41522887 is $1000
>>> my_savings.add_interest()
>>> my_savings.check_balance()
The balance of account number 41522887 is $1055.00
```

The second subclass, `CurrentAccount`, has a similar structure:

```
class CurrentAccount(BankAccount):
    """ A class representing a current (checking) account. """
    def __init__(self, customer, account_number, annual_fee,
                 transaction_limit, balance=0):
        """ Initialize the current account. """

        self.annual_fee = annual_fee
        self.transaction_limit = transaction_limit
        super().__init__(customer, account_number, balance)

    def withdraw(self, amount):
        """
        Withdraw amount if sufficient funds exist in the account and amount
        is less than the single transaction limit.

        """
        if amount <= 0:
            print('Invalid withdrawal amount:', amount)
            return

        if amount > self.balance:
            print('Insufficient funds')
            return

        if amount > self.transaction_limit:
            print('{0:s}{1:.2f} exceeds the single transaction limit of'
                  ' {0:s}{2:.2f}'.format(self.currency, amount,
                                         self.transaction_limit))
            return
```

[17] The built-in function `super()` called in this way creates a "proxy" object that delegates method calls to the parent class (in this case, `BankAccount`).

```
        self.balance -= amount

    def apply_annual_fee(self):
        """ Deduct the annual fee from the account balance. """

        self.balance = max(0., self.balance - self.annual_fee)
```

Note what happens if we call `withdraw` on a `CurrentAccount` object:

```
>>> my_current = CurrentAccount('Alison Wicks', 78300991, 20., 200.)
>>> my_current.withdraw(220)
Insufficient Funds

>>> my_current.deposit(750)
>>> my_current.check_balance()
The balance of account number 78300991 is $750.00

>>> my_current.withdraw(220)
$220.00 exceeds the transaction limit of $200.00
```

The `withdraw` method called is that of the `CurrentAccount` class, as this method overrides that of the same name in the base class, `BankAccount`.

Example E4.17 A simple model of a polymer in solution treats it as a sequence of randomly oriented segments; that is, one for which there is no correlation between the orientation of one segment and any other (this is the so-called *random-flight* model).

We will define a class, `Polymer`, to describe such a polymer, in which the segment positions are held in a list of `(x,y,z)` tuples. A `Polymer` object will be initialized with the values `N` and `a`, the number of segments and the segment length respectively. The initialization method calls a `make_polymer` method to populate the segment positions list.

The `Polymer` object will also calculate the end-to-end distance, R, and will implement a method `calc_Rg` to calculate and return the polymer's *radius of gyration*, defined as

$$R_{\mathrm{g}} = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(\mathbf{r}_i - \mathbf{r}_{\mathrm{CM}})^2}$$

Listing 4.7 Polymer class

```
# polymer.py

import math
import random

class Polymer:
    """ A class representing a random-flight polymer in solution. """

    def __ init __(self, N, a):
        """
        Initialize a Polymer object with N segments, each of length a.
```

```
        """

        self.N, self.a = N, a
        # self.xyz holds the segment position vectors as tuples
        self.xyz = [(None, None, None)] * N
        # End-to-end vector
        self.R = None
        # Make our polymer by assigning segment positions
        self.make_polymer()

    def make_polymer(self):
        """
        Calculate the segment positions, center of mass and end-to-end
        distance for a random-flight polymer.

        """

        # Start our polymer off at the origin, (0,0,0).
        self.xyz[0] = x, y, z = cx, cy, cz = 0., 0., 0.
        for i in range(1, self.N):
❶           # Pick a random orientation for the next segment.
            theta = math.acos(2 * random.random() - 1)
            phi = random.random() * 2. * math.pi
            # Add on the corresponding displacement vector for this segment.
            x += self.a * math.sin(theta) * math.cos(phi)
            y += self.a * math.sin(theta) * math.sin(phi)
            z += self.a * math.cos(theta)
            # Store it, and update or center of mass sum.
            self.xyz[i] = x, y, z
            cx, cy, cz = cx + x, cy + y, cz + z
❷       # Calculate the position of the center of mass.
        cx, cy, cz = cx / self.N, cy / self.N, cz / self.N
        # The end-to-end vector is the position of the last
        # segment, since we started at the origin.
        self.R = x, y, z

        # Finally, re-center our polymer on the center of mass.
        for i in range(self.N):
            self.xyz[i] = self.xyz[i][0]-cx,self.xyz[i][1]-cy,self.xyz[i][2]-cz
    def calc_Rg(self):
        """
        Calculates and returns the radius of gyration, Rg. The polymer
        segment positions are already given relative to the center of
        mass, so this is just the rms position of the segments.

        """

        self.Rg = 0.
        for x,y,z in self.xyz:
            self.Rg += x**2 + y**2 + z**2
        self.Rg = math.sqrt(self.Rg / self.N)
        return self.Rg
```

❶ One way to pick the location of the next segment is to pick a random point on the surface of the unit sphere and use the corresponding pair of angles in the spherical polar coordinate system, θ and ϕ (where $0 \le \theta < \pi$ and $0 \le \phi < 2\pi$) to set the displacement

from the previous segment's position as

$$\Delta x = a \sin\theta \cos\phi$$
$$\Delta y = a \sin\theta \sin\phi$$
$$\Delta z = a \cos\theta$$

❷ We calculate the position of the polymer's center of mass, $\mathbf{r}_{\mathrm{CM}}$, and then shift the origin of the polymer's segment coordinates so that they are measured relative to this point (that is, the segment coordinates have their origin at the polymer center of mass).

We can test the `Polymer` class by importing it in the Python shell:

```
>>> from polymer import Polymer
>>> polymer = Polymer(1000, 0.5)    # A polymer with 1000 segments of length 0.5
>>> polymer.R           # End-to-end vector
(5.631332375722011, 9.408046667059947, -1.3047608473668109)
>>> polymer.calc_Rg()   # Radius of gyration
5.183761585363432
```

Let's now compare the distribition of the end-to-end distances with the theoretically predicted probability density function:

$$P(R) = 4\pi R^2 \left(\frac{3}{2\pi\langle r^2\rangle}\right)^{3/2} \exp\left(-\frac{3R^2}{2\langle r^2\rangle}\right),$$

where the mean square position of the segments is $\langle r^2\rangle = Na^2$

Listing 4.8 The distribution of random flight polymers

```
# eg4-c-ii-polymer-a.py
# Compare the observed distribution of end-to-end distances for Np random-
# flight polymers with the predicted probability distribution function.

import pylab
from polymer import Polymer
pi = pylab.pi

# Calculate R for Np polymers
Np = 3000
# Each polymer consists of N segments of length a
N, a = 1000, 1.
R = [None] * Np
for i in range(Np):
    polymer = Polymer(N, a)
    Rx, Ry, Rz = polymer.R
    R[i] = pylab.sqrt(Rx**2 + Ry**2 + Rz**2)
    # Output a progress indicator every 100 polymers
    if not (i+1) % 100:
        print(i+1, '/', Np)

# Plot the distribution of Rx as a normalized histogram
# using 50 bins
pylab.hist(R, 50, normed=1)

# Plot the theoretical probability distribution, Pr, as a function of r
r = pylab.linspace(0,200,1000)
```

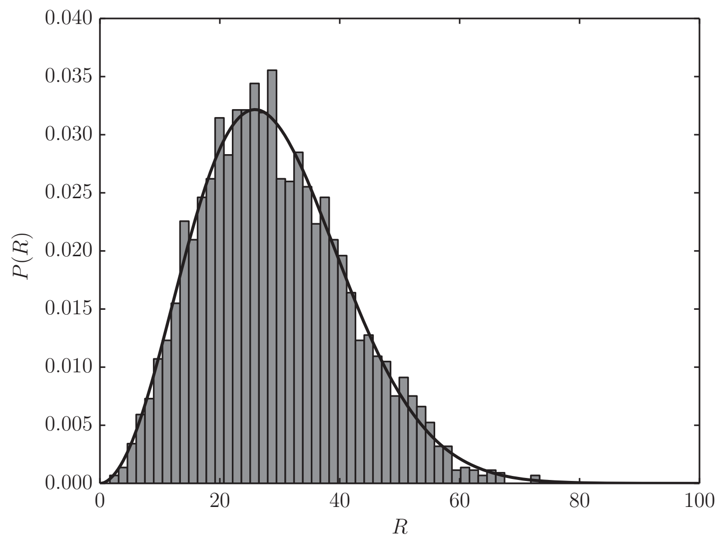


Figure 4.4 Distribution of the end-to-end distances, R, of random flight-polymers with $N = 1,000, a = 1$.

```
msr = N * a**2
Pr = 4.*pi*r**2 * (2 * pi * msr / 3)**-1.5 * pylab.exp(-3*r**2 / 2 / msr)
pylab.plot(r, Pr, lw=2, c='r')
pylab.xlabel('R')
pylab.ylabel('P(R)')
pylab.show()
```

The earlier mentioned program produces a plot that typically looks like Figure 4.4, suggesting agreement with theory.

4.6.4 Exercises

Problems

P4.6.1 a. Modify the base `BankAccount` class to verify that the account number passed to its `__init__` constructor conforms to the Luhn algorithm described in Exercise P2.5.3.

b. Modify the `CurrentAccount` class to implement a free overdraft. The limit should be set in the `__init__` constructor; withdrawals should be allowed to within the limit.

P4.6.2 Add a method, `save_svg` to the `Polymer` class of Example E4.17 to save an image of its polymer as an SVG file. Refer to Exercise P4.4.3 for a template of an SVG file.

P4.6.3 Write a program to create an image of a constellation using the data from the Yale Bright Star Catalog (http://tdc-www.harvard.edu/catalogs/bsc5.html).

Create a class, `Star`, to represent a star with attributes for its name, magnitude and position in the sky, parsed from the file `bsc5.dat` which forms part of the catalog. Implement a method for this class which converts the star's position on the celestial sphere as (Right Ascension: α, Declination: δ) to a point in a plane, (x, y), for example using the orthographic projection about a central point (α_0, δ_0):

$$\begin{aligned}\Delta\alpha &= \alpha - \alpha_0 \\ x &= \cos\delta \sin\Delta\alpha \\ y &= \sin\delta\cos\delta_0 - \cos\delta\cos\Delta\alpha\sin\delta_0\end{aligned}$$

Suitably scaled projected, star positions can be output to an SVG image as `circles` (with a larger radius for brighter stars). For example, the line

```
<circle cx="200" cy="150" r="5" stroke="none" fill="#ffffff"/>
```

represents a white circle of radius 5 pixels, center on the canvas at (200, 150).

Hint: you will need to convert the right ascension from (hr, min, sec) and the declination from (deg, min, sec) to radians. Use the data corresponding to "equinox J2000, epoch 2000.0" in each line of `bsc5.dat`. Let the user select the constellation from the command line using its three-letter abbreviation (e.g., 'Ori' for Orion): this is given as part of the star name in the catalog. Don't forget that star magnitudes are *smaller* for *brighter* stars. If you are using the orthographic projection suggested, choose (α_0, δ_0) to be the mean of (α, δ) for stars in the constellation.

P4.6.4 Design and implement a class, `Experiment`, to read in and store a simple series of (x, y) data as `pylab` (i.e., NumPy) arrays from a text file. Include in your class methods for transforming the data series by some simple function (e.g., $x' = \ln x$, $y' = 1/y$) and to perform a linear leastsquares regression on the transformed data (returning the gradient and intercept of the best-fit line, $y'_{\mathrm{fit}} = mx' + c$). NumPy provides methods for performing linear regression (see Section 6.5.3), but for this exercise the following equations can be implemented directly:

$$\begin{aligned}m &= \frac{\overline{xy} - \bar{x}\bar{y}}{\overline{x^2} - \bar{x}^2} \\ c &= \bar{y} - m\bar{x}\end{aligned}$$

where the bar notation, $\bar{\cdot}$, denotes the arithmetic mean of the quantity under it. (*Hint:* use `pylab.mean(arr)` to return the mean of array `arr`.)

Chloroacetic acid is an important compound in the synthetic production of phamaceuticals, pesticides and fuels. At high concentration under strong alkaline conditions its hydrolysis may be considered as the following reaction:

$$ClCH_2COO^- + OH^- \rightleftharpoons HOCH_2COO^- + Cl^-.$$

Data giving the concentration of $ClCH_2COO^-$, c (in M), as a function of time, t (in s), are provided for this reaction carried out in excess alkalai at five different temperatures in the data files `caa-T.txt` (`T = 40, 50, 60, 70, 80` in °C): these may be obtained

from scipython.com/ex/ade. The reaction is known to be second order and so obeys the integrated rate law

$$\frac{1}{c} = \frac{1}{c_0} + kt$$

where k is the effective rate constant and c_0 the initial ($t = 0$) concentration of chloroacetic acid.

Use your `Experiment` class to interpret these data by linear regression of $1/c$ against t, determining $m(\equiv k)$ for each temperature. Then, for each value of k, determine the activation energy of the reaction through a second linear regression of $\ln k$ against $1/T$ in accordance with the Arrhenius law:

$$k = Ae^{-E_a/RT} \quad \Rightarrow \quad \ln k = \ln A - \frac{E_a}{RT},$$

where $R = 8.314\ \mathrm{J\,K^{-1}\,mol^{-1}}$ is the gas constant. Note: the temperature must be in Kelvin.

5 IPython and IPython Notebook

The IPython shell and the related interactive, browser-based IPython Notebook provide a powerful interface to the Python language. IPython has several advantages over the native Python shell, including easy interaction with the operating system, introspection and tab completion. IPython Notebook increasingly is being adopted by scientists to share their data and the code they write to analyze it in a standardized manner that aids reproducibility and visualization.

5.1 IPython

5.1.1 Installing IPython

Comprehensive details on installing IPython are available at the IPython website: see http://ipython.org/install.html, but a summary is provided here.

IPython is included in the Continuum Anaconda and Enthought Canopy Python distributions. To update to the current version within Anaconda, use the `conda` package manager:

```
conda update conda
conda update ipython
```

With Canopy, use

```
enpkg ipython
```

If you are not using these distributions but already have Python installed, there are several alternative options. If you have the `pip` package manager:

```
pip install ipython
pip install "ipython[notebook]"
```

It is also possible to manually download the latest IPython version from the github repository at https://github.com/ipython/ipython/releases and compile and install from its top-level source directory with

```
python setup.py install
```

5.1.2 Using the IPython shell

To start an interactive IPython shell session from the command line, simply type `ipython`. You should be greeted with a message similar to this one:

```
Python 3.3.5 |Anaconda 2.0.1 (x86_64)| (default, Mar 10 2014, 11:22:25)
Type "copyright", "credits" or "license" for more information.

IPython 2.1.0 -- An enhanced Interactive Python.
Anaconda is brought to you by Continuum Analytics.
Please check out: http://continuum.io/thanks and https://binstar.org
?         -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help      -> Python's own help system.
object?   -> Details about 'object', use 'object??' for extra details.

In [1]:
```

(The precise details of this message will depend on the setup of your system.) The prompt `In [1]:` is where you type your Python statements and replaces the native Python `>>>` shell prompt. The counter in square brackets increments with each Python statement or code block. For example,

```
In [1]: 4+5
Out[1]: 9
In [2]: print(1)
1
In [3]: for i in range(4):
   ...:     print(i, end='')
   ...:
0123
In [4]:
```

To exit the IPython shell, type `quit` or `exit`. Unlike with the native Python shell, no parentheses are required.[1]

Help commands

As listed in the welcome message, there are various helpful commands to obtain information about using IPython:

- Typing a single '`?`' outputs an overview of the usage of IPython's main features (page down with the space bar or `f`; page back up with `b`; exit the help page with `q`).
- `%quickref` provides a brief reference summary of each of the main IPython commands and "magics" (see Section 5.1.3).
- `help()` or `help(`*`object`*`)` invokes Python's own help system (interactively or for *`object`* if specified).
- Typing one question mark after an object name provides information about that object: see below.

[1] Some find this alone a good reason to use IPython.

Possibly the most frequently used help functionality provided by IPython is the *introspection* provided by the `object?` syntax. For example,

```
In [4]: a = [5, 6]
In [5]: a?
Type:        list
String form: [5, 6]
Length:      2
Docstring:
list() -> new empty list
list(iterable) -> new list initialized from iterable's items
```

Here, the command `a?` gives details about the object `a`: its string representation (which would be produced by, for example, `print(a)`), its length (equivalent to `len(a)` and the docstring associated with the class of which it is an instance: since `a` is a list, this provides brief details of how to instantiate a `list` object.[2]

The `?` syntax is particularly useful as a reminder of the arguments that a function or method takes. For example,

```
In [6]: import pylab
In [7]: pylab.linspace?

String form: <function linspace at 0x10432d560>
File:        /Users/christian/anaconda/envs/py33/lib/python3.3/site-packages/numpy/
             core/function_base.py
Definition:  pylab.linspace(start, stop, num=50, endpoint=True, retstep=False)
Docstring:
Return evenly spaced numbers over a specified interval.

Returns `num` evenly spaced samples, calculated over the
interval [`start`, `stop` ].

The endpoint of the interval can optionally be excluded.

Parameters
----------
start : scalar
    The starting value of the sequence.
stop : scalar
    The end value of the sequence, unless `endpoint` is set to False.
    In that case, the sequence consists of all but the last of ``num + 1``
    evenly spaced samples, so that `stop` is excluded. Note that the step
    size changes when `endpoint` is False.
num : int, optional
    Number of samples to generate. Default is 50.
endpoint : bool, optional
    If True, `stop` is the last sample. Otherwise, it is not included.
    Default is True.
retstep : bool, optional
    If True, return (`samples`, `step`), where `step` is the spacing
    between samples.
```

[2] This is what is meant by introspection: Python is able to inspect its own objects and provide information about them.

```
Returns
-------
...

See Also
--------
...

Examples
--------
...
```

For some objects, the syntax `object??` returns more advanced information such as the location and details of its source code.

Tab completion

Just as with many command line shells, IPython supports tab completion: start typing the name of an object or keyword, press the <TAB> key, and it will autocomplete it for you or provide a list of options if more than one possibility exists. For example,

```
In [8]: w<TAB>
%%writefile  %who         %who_ls      %whos        while        with

In [8]: w
```

If you resume typing until the word becomes unambiguous (e.g., add the letters `hi`) and then press <TAB> again: it will be autocompleted to `while`. The options with percent signs in front of them are "magic functions," described in Section 5.1.3.

History

You may already have used the native Python shell's command history functionality (pressing the up and down arrows through previous statements typed during your current session). IPython stores both the commands you enter and the output they produce in the special variables `In` and `Out` (these are, in fact, a list and a dictionary respectively, and correspond to the prompts at the beginning of each input and output). For example,

```
In [9]: d = {'C': 'Cador', 'G': 'Galahad', 'T': 'Tristan', 'A': 'Arthur'}
In [10]: for a in 'ACGT':
   ....:     print(d[a])
   ....:
Arthur
Cador
Galahad
Tristan
In [11]: d = {'C': 'Cytosine', 'G': 'Guanine', 'T': 'Thymine', 'A': 'Adenine'}
❶ In [12]: In[10]
Out[12]: "for a in 'ACGT':\n    print(d[a])\n    "
❷ In [13]: exec(In[10])
Adenine
Cytosine
Guanine
Thymine
```

❶ Note that `In[10]` simply holds the string version of the Python statement (here a `for` loop) that was entered at index 10.

❷ To actually execute the statement (with the *current* dictionary `d`), we must send it to Python's `exec` built-in (see also the `%rerun` magic, Section 5.1.3).

There are a couple of further shortcuts: the alias `_iN` is the same as `In[N]`, `_N` is the same as `Out[N]`, and the two most recent outputs are returned by the variables _ and __ respectively.

To view the contents of the history, use the `%history` or `%hist` magic function. By default only the entered statements are output; it is often more useful to output the line numbers as well, which is acheived using the `-n` option:

```
In [14]: %history -n
   1: 4+5
   2: print(1)
   3:
for i in range(4):
    print(i)
   4: a = [5, 6]
   5: a?
   6: import pylab
   7: pylab.linspace?
   8: d = {'C': 'Cador', 'G': 'Galahad', 'T': 'Tristan', 'A': 'Arthur'}
  10:
for a in 'ACGT':
    print(d[a])
  11: d = {'C': 'Cytosine', 'G': 'Guanine', 'T': 'Thymine', 'A': 'Adenine'}
  12: In[10]
  13: exec(In[10])
  14: %history -n
```

To output a specific line or range of lines, refer to them by number and/or number range when calling `%history`:

```
In [15]: %history 4
a = [5, 6]

In [16]: %history -n 2-5
   2: print(1)
   3:
for i in range(4):
    print(i)
   4: a = [5, 6]
   5: a?

In [17]: %history -n 1-3 7 12-14
   1: 4+5
   2: print(1)
   3:
for i in range(4):
    print(i)
   7: pylab.linspace?
  12: In[10]
  13: exec(In[10])
  14: %history -n
```

This syntax is also used by several other IPython magic functions (see the following section). The `%history` function can also take an additional option: `-o` displays the output as well as the input.

Pressing `CTRL-R` brings up a prompt, the somewhat cryptic `(reverse-i-search)``':`, from which you can search within your command history.[3]

Interacting with the operating system

IPython makes it easy to execute operating system commands from within your shell session: any statement preceded by an exclamation mark, `!`, is sent to the operating system command line (the "system shell") instead of being executed as a Python statement. For example, you can delete files, list directory contents and even execute other programs and scripts:

```
In [11]: !pwd               # return the current working directory
/Users/christian/research
In [12]: !ls                # list the files in this directory
Meetings          Papers            code              books
databases         temp-file
In [13]: !rm temp-file      # delete temp-file

In [14]: !ls
Meetings          Papers            code              books
databases
```

Note that, for technical reasons,[4] the `cd` (Unix-like systems) and `chdir` (Windows) commands must be executed as IPython magic functions:

```
In [15]: %cd /       # Change into root directory
In [16]: !ls
Applications      Volumes           usr               Library
bin               net               Network           cores
opt               www               System            dev
private           sbin              Users             home
In [17]: %cd ~/temp   # Change directory to temp within user's home directory
In [18]: !ls
output.txt        test.py           readme.txt        utils
zigzag.py
```

If you use Windows and want to include a drive letter (such as `C:`) in the directory path you should enclose the path in quotes: `%cd 'C:\My Documents'`.

Help, via `!`*`command`*`?`, and tab completion, as described in Section 5.1.2, work within operating system commands.

You can pass the values of Python variables to operating system commands by prefixing the variable name with a dollar sign, `$`:

```
In [19]: python_script = 'zigzag.py'
In [20]: !ls $python_script
```

[3] This functionality may be familiar to users of the bash shell.

[4] System commands executed via the `!`*`command`* method spawn their own shell, which is discarded immediately afterward; changing a directory occurs only in this spawned shell and is not reflected in the one running IPython.

```
zigzag.py
In [21]: text_files = '*.txt'
❶ In [22]: text_file_list = !ls $text_files
In [23]: text_file_list
output.txt   readme.txt
In [24]: readme_file = text_file_list[1]
In [25]: !cat $readme_file
This is the file readme.txt
Each line of the file appears as an item
in a list when returned from !cat readme.txt

❷ In [26]: readme_lines = !cat $readme_file

In [27]: readme_lines
Out[28]:
['This is the file readme.txt',
 'Each line of the file appears as an item',
 'in a list when returned from !cat readme.txt']
```

❶ Note that the output of a system command can be assigned to a Python variable, here a list of the `.txt` files in the current directory.
❷ The `cat` system command returns the contents of the text file; IPython splits this output on the newline character and assigns the resulting list to `readme_lines`. See also Section 5.1.3

5.1.3 IPython magic functions

IPython provides many "magic" functions (or simply *magics*, those commands prefixed with `%`) to speed up coding and experimenting within the IPython shell. Some of the more useful ones are described in this section; for more advanced information the reader is referred to the IPython documentation.[5] IPython makes a distinction between *line magics*: those whose arguments are given on a single line, and *cell magics* (prefixed by two percent signs, `%%`): those which act on a series of Python commands. An example is given in Section 5.1.3 where we describe the `%%timeit` cell magic.

A list of currently available magic functions can be obtained by typing `%lsmagic`.

The magic function `%automagic` toggles the "automagic" setting: its default is `ON` meaning that typing the name of a magic function without the `%` will also execute that function, unless you have bound the name as a Python identifier (variable name) to some object. The same principle applies to system commands:

```
In [x]: ls
output.txt        test.py           readme.txt        utils
zigzag.py
In [x]: ls = 0
In [x]: ls          # Now ls is an integer; !ls will still work
Out[x]: 0
```

Table 5.1 summarizes some useful IPython magics; the following subsections explain more fully the less straightforward ones.

[5] http://ipython.org/documentation.html.

Table 5.1 Useful IPython line magics

Magic	Description
`%alias`	Create an alias to a system command.
`%alias_magic`	Create an alias to an existing IPython magic.
`%bookmark`	Interact with IPython's directory bookmarking system.
`%cd`	Change the current working directory.
`%dhist`	Output a list of visited directories.
`%edit`	Create or edit Python code within a text editor and then execute it.
`%env`	List the system environment variables, such as `$HOME`.
`%history`	List the input history for this IPython session.
`%load`	Read in code from a provided file and make it available for editing.
`%macro`	Define a named macro from previous input for future reexecution.
`%paste`	Paste input from the clipboard: use this in preference to, for example, `CTRL-V`, to handle code indenting properly.
`%pylab`	Activate the `pylab` library within the current session for interactive plotting.
`%recall`	Place one or more input lines from the command history at the current input prompt.
`%rerun`	Reexecute previous input from the numbered command history.
`%reset`	Reset the namespace for the current IPython session.
`%run`	Execute a named file as a Python script within the current session.
`%save`	Save a set of input lines or macro (defined with `%macro`) to a file with a given name.
`%sx` or `!!`	Shell execute: run a given shell command and store its output.
`%timeit`	Time the execution of a provided Python statement.
`%who`	Output all the currently defined variables.
`%who_ls`	As for `%who`, but return the variable names as a list of strings.
`%whos`	As for `%who`, but provides more information about each variable.

Aliases and bookmarks

A system shell command can be given an *alias*: a shortcut for a shell command that can be called as its own magic. For example, on Unix-like systems we could define the following alias to list only the directories on the current path:

```
In [x]: %alias lstdir ls -d */
In [x]: %lstdir
Meetings/          Papers/          code/          books/
databases/
```

Now typing `%lstdir` has the same effect as `!ls -d */`. If `%automagic` is `ON` this alias can also simply be called with `lstdir`.

The magic `%alias_magic` provides a similar functionality for IPython magics. For example, if you want to use `%h` as an alias to `%history`, type:

```
In [x]: %alias_magic h history
```

When working on larger projects it is often necessary to switch between different directories. IPython has a simple system for maintaining a list of bookmarks which act as shortcuts to different directories. The syntax for this magic function is

`%bookmark <name> [directory]`

If `[directory]` is omitted, it defaults to the current working directory.

```
In [x]: %bookmark py ~/research/code/python
In [x]: %bookmark www /srv/websites
In [x]: %cd py
/Users/christian/research/code/python
```

It may happen that a directory with the same name as your bookmark is within the current working directory. In that case, this directory takes precedence and you must use `%cd -b <name>` to refer to the bookmark.

A few more useful commands include:

- `%bookmark -l`: list all bookmarks
- `%bookmark -d <name>`: remove bookmark `<name>`
- `%bookmark -r`: remove all bookmarks

Timing code execution

The IPython magic `%timeit <statement>` times the execution of the *single-line* statement `<statement>`. The statement is executed N times in a loop, and each loop is repeated R times. N is a suitable, usually large, number chosen by IPython to yield meaningful results and R is, by default, 3. The average time per loop for the best of the R repetitions is reported. For example, to profile the sorting of a random arrangement of the numbers 1–100:

```
In [x]: import random
In [x]: numbers = list(range(1,101))
In [x]: random.shuffle(numbers)
In [x]: %timeit sorted(numbers)
100000 loops, best of 3: 13.2 µs per loop
```

Obviously the execution time will depend on the system (processor speed, memory, etc.). The aim of repeating the execution many times is to allow for variations in speed due to other processes running on the system. You can select N and R explicitly by passing values to the options `-n` and `-r` respectively:

```
In [x]: %timeit -n 10000 -r 5 sorted(numbers)
10000 loops, best of 5: 11.2 µs per loop
```

The cell magic, `%%timeit` enables one to time a *multiline block* of code. For example, a naive algorithm to find the factors of an integer `n` can be examined with

```
In [x]: n = 150
In [x]: %%timeit
factors = set()
for i in range(1, n+1):
    if not n % i:
        factors.add(n // i)
   ....:

100000 loops, best of 3: 16.3 µs per loop
```

Recalling and rerunning code

To reexecute one or more lines from your IPython history, use `%rerun` with a line number or range of line numbers:

```
In [1]: import math
In [2]: angles = [0, 30, 60, 90]
In [3]: for angle in angles:
    sine_angle = math.sin(math.radians(angle))
    print('sin({:3d}) = {:8.5f}'.format(angle, sine_angle))
    .....:
sin(  0) =  0.00000
sin( 30) =  0.50000
sin( 45) =  0.70711
sin( 60) =  0.86603
sin( 90) =  1.00000

In [4]: angles = [15, 45, 75]
In [5]: %rerun 3
=== Executing: ===
for angle in angles:
    sine_angle = math.sin(math.radians(angle))
    print('sin({:3d}) = {:8.5f}'.format(angle, sine_angle))

=== Output: ===
sin( 15) =  0.25882
sin( 45) =  0.70711
sin( 75) =  0.96593

In [6]: %rerun 2-3
=== Executing: ===
angles = [0, 30, 45, 60, 90]
for angle in angles:
    sine_angle = math.sin(math.radians(angle))
    print('sin({:3d}) = {:8.5f}'.format(angle, sine_angle))

=== Output: ===
sin(  0) =  0.00000
sin( 30) =  0.50000
sin( 45) =  0.70711
sin( 60) =  0.86603
sin( 90) =  1.00000
```

The similar magic function `%recall` places the requested lines at the command prompt but does not execute them until you press Enter, allowing you to modify them first if you need to.

If you find yourself reexecuting a series of statements frequently, you can define a named macro to invoke them. Specify line numbers as before:

```
In [7]: %macro sines 3
Macro `sines` created. To execute, type its name (without quotes).
=== Macro contents: ===
for angle in angles:
    sine_angle = math.sin(math.radians(angle))
    print('sin({:3d}) = {:8.5f}'.format(angle, sine_angle))
```

```
In [8]: angles = [-45, -30, 0, 30, 45]
In [9]: sines
sin(-45) = -0.70711
sin(-30) = -0.50000
sin(  0) =  0.00000
sin( 30) =  0.50000
sin( 45) =  0.70711
```

Loading, executing and saving code

To load code from an external file into the current IPython session, use

```
%load <filename>
```

If you want only certain lines from the input file, specify them after the `-r` option. This magic enters the lines at the command prompt, so they can be edited before being executed.

To load *and execute* code from a file, use

```
%run <filename>
```

Pass any command line options after *filename*; by default IPython treats them the same way that the system shell would. There are a few additional options to `%run`:

- `-i`: Run the script in the current IPython namespace instead of an empty one (i.e., the program will have access to variables defined in the current IPython session);
- `-e`: Ignore `sys.exit()` calls and `SystemExit` exceptions;
- `-t`: Output timing information at the end of execution (pass an integer to the additional option `-N` to repeat execution that number of times).

For example, to run `my_script.py` 10 times from within IPython with timing information:

```
In [x]: %run -t -N10 my_script.py
```

To save a range of input lines or a macro to a file, use `%save`. Line numbers are specified using the same syntax as `%history`. A `.py` extension is added if you don't add it yourself, and confirmation is sought before overwriting an existing file. For example,

```
In [x]: %save sines1 1 8 3
The following commands were written to file 'sines1.py':
import math
angles = [-45, -30, 0, 30, 45]
for angle in angles:
    print('sin({:3d}) = {:8.5f}'.format(angle, math.sin(math.radians(angle))))

In [x]: %save sines2 1-3
The following commands were written to file 'sines2.py':
import math
angles = [0, 30, 60, 90]
for angle in angles:
    print('sin({:3d}) = {:8.5f}'.format(angle, math.sin(math.radians(angle))))
```

Finally, to *append* to a file instead of overwriting it, use the `-a` option:

```
%save -a <filename> <line numbers>
```

Capturing the output of a shell command

The IPython magic `%sx` *`command`*, equivalent to `!!`*`command`* executes the shell command *`command`* and returns the resulting output as a list (split into semantically useful parts on the new line character so there is one item per line). This list can be assigned to a variable to be manipulated later. For example,

```
In [x]: current_working_directory = %sx pwd
In [x]: current_working_directory
['/Users/christian/temp']
In [x]: filenames = %sx ls
In [x]: filenames
Out[x]:
['output.txt',
 'test.py',
 'readme.txt',
 'utils',
 'zigzag.py']
```

Here, `filenames` is a list of individual filenames.

The returned object is actually an `IPython.utils.text.SList` string list object. Among the useful additional features provided by `SList` are a native method for splitting each string into fields delimited by whitespace: `fields`; for sorting on those fields: `sort`; and for searching within the string list: `grep`. For example,

```
In [x]: files = %sx ls -l
In [x]: files
['total 8',
 '-rw-r--r--  1 christian  staff     93  5 Nov 16:30 output.txt',
 '-rw-r--r--  1 christian  staff  23258  5 Nov 16:31 readme.txt',
 '-rw-r--r--  1 christian  staff    218  5 Nov 16:32 test.py',
 'drwxr-xr-x  2 christian  staff     68  5 Nov 16:32 utils',
 '-rw-r--r--  1 christian  staff    365  5 Nov 16:20 zigzag.py']
In [x]: del files[0]    # strip non-file line 'total 8'
In [x]: files.fields()
Out[x]:
[['-rw-r--r--', '1', 'christian', 'staff', '93', '5', 'Nov', '16:30', 'output.txt'],
 ['-rw-r--r--', '1', 'christian', 'staff', '23258', '5', 'Nov', '16:31', 'readme.txt'],
 ...
 ['-rw-r--r--', '1', 'christian', 'staff', '365', '5', 'Nov', '16:20', 'zigzag.py']]

In [x]: ['{} last modified at {} on {} {}'.format(f[8], f[7], f[5], f[6])
                                    for f in files.fields()]
Out[x]:
['output.txt last modified at 16:30 on 5 Nov',
 'readme.txt last modified at 16:31 on 5 Nov',
 'test.py last modified at 16:32 on 5 Nov',
 'utils last modified at 16:32 on 5 Nov',
 'zigzag.py last modified at 16:20 on 5 Nov']
```

The `fields` method can also take arguments specifying the indexes of the fields to output; if more than one index is given the fields are joined by spaces:

```
In [x]: files.fields(0)     # First field in each line of files
Out[x]: ['-rw-r--r--', '-rw-r--r--', '-rw-r--r--', 'drwxr-xr-x', '-rw-r--r--']
```

```
In [x]: files.fields(-1)    # Last field in each line of files
Out[x]: ['output.txt', 'readme.txt', 'test.py', 'utils', 'zigzag.py']

In [x]: files.fields(8,7,5,6)
Out[x]:
['output.txt 16:30 5 Nov',
 'readme.txt 16:31 5 Nov',
 'test.py 16:32 5 Nov',
 'utils 16:32 5 Nov',
 'zigzag.py 16:20 5 Nov']
```

The `sort` method provided by `SList` objects can sort by a given field, optionally converting the field from a string to a number if required (so that, for example, `10 > 9`). Note that this method returns a new `SList` object.

```
In [x]: files.sort(4)      # Sort alphanumerically by size (not useful)
Out[x]:
['-rw-r--r--  1 christian  staff    218  5 Nov 16:32 test.py',
 '-rw-r--r--  1 christian  staff  23258  5 Nov 16:31 readme.txt',
 '-rw-r--r--  1 christian  staff    365  5 Nov 16:20 zigzag.py',
 'drwxr-xr-x  2 christian  staff     68  5 Nov 16:32 utils',
 '-rw-r--r--  1 christian  staff     93  5 Nov 16:30 output.txt']

In [x]: files.sort(4, nums=True)      # Sort numerically by size (useful)
Out[x]:
['drwxr-xr-x  2 christian  staff     68  5 Nov 16:32 utils',
 '-rw-r--r--  1 christian  staff     93  5 Nov 16:30 output.txt',
 '-rw-r--r--  1 christian  staff    218  5 Nov 16:32 test.py',
 '-rw-r--r--  1 christian  staff    365  5 Nov 16:20 zigzag.py',
 '-rw-r--r--  1 christian  staff  23258  5 Nov 16:31 readme.txt']
```

The `grep` method returns items from the `SList` containing a given string;[6] to search for a string in a given field only, use the `field` argument:

```
In [x]: files.grep('txt')            # Search for lines containing 'txt'
Out[x]:
['-rw-r--r--  1 christian  staff     93  5 Nov 16:30 output.txt',
 '-rw-r--r--  1 christian  staff  23258  5 Nov 16:31 readme.txt']

In [x]: files.grep('16:32', field=7)    # Search file files created at 16:32
Out[x]:
['-rw-r--r--  1 christian  staff    218  5 Nov 16:32 test.py',
 'drwxr-xr-x  2 christian  staff     68  5 Nov 16:32 utils']
```

Example E5.1 RNA encodes the amino acids of a peptide as a sequence of *codons*, with each codon consisting of three nucleotides chosen from the 'alphabet': U (uracil), C (cytosine), A (adenine) and G (guanine).

The Python script, `codon_lookup.py`, available at scipython.com/eg/aab, creates a dictionary, `codon_table`, mapping codons to amino acids where each amino acid is identified by its one-letter abbreviation (e.g., `R` = arginine). The stop codons, signaling termination of RNA translation, are identified with the single asterisk character, `*`.

[6] In fact, its name implies it will match *regular expressions* as well, but we will not expand on this here.

The codon AUG signals the start of translation within a nucleotide sequence as well as coding for the amino acid methionine.

This script can be executed within IPython with `%run codon_lookup.py` (or loaded and then executed with `%load codon_lookup.py` followed by pressing Enter:

```
In [x]: %run codon_lookup.py
In [x]: codon_table
Out[x]:
{'GCG': 'A',
 'UAA': '*',
 'GGU': 'G',
 'UCU': 'S',
     ...
 'ACA': 'T',
 'ACC': 'T'}
```

Let's define a function to translate an RNA sequence. Type `%edit` and enter the following code in the editor that appears.

```
def translate_rna(seq):
    start = seq.find('AUG')
    peptide = []
    i = start
    while i < len(seq)-2:
        codon = seq[i:i+3]
        a = codon_table[codon]
        if a == '*':
            break
        i += 3
        peptide.append(a)
    return ''.join(peptide)
```

When you exit the editor it will be executed, defining the function, `translate_rna`:

```
IPython will make a temporary file named: /var/folders/fj/yv29fhm91v7_6g
7sqsy1z2940000gp/T/ipython_edit_thunq9/ipython_edit_dltv_i.py
Editing... done. Executing edited code...
Out[x]: "def translate_rna(seq):\n    start = seq.find('AUG')\n
peptide = []\
n    i = start\n    while i < len(seq)-2:\n        codon = seq[i:i+3]\n        a
 = codon_table[codon]\n        if a == '*':\n            break\n        i += 3\n
       peptide.append(a)\n    return ''.join(peptide)\n"
```

Now feed the function an RNA sequence to translate:

```
In[x]: seq = 'CAGCAGCUCAUACAGCAGGUAAUGUCUGGUCUCGUCCCCGGAUGUCGCUACCCACGAG
ACCCGUAUCCUACUUUCUGGGGAGCCUUUACACGGCGGUCCACGUUUUUCGCUACCGUCGUUUUCCCGGUGC
CAUAGAUGAAUGUU'
In [x]: translate_rna(seq)
Out[x]: 'MSGLVPGCRYPRDPYPTFWGAFTRRSTFFATVVFPVP'
```

To read in a list of RNA sequences (one per line) from a text file, `seqs.txt`, and translate them, one could use `%sx` with the system command `cat` (or, on Windows, the command `type`):

```
In [x]: seqs = %sx cat seqs.txt
In [x]: for seq in seqs:
```

```
   ...:     print(translate_rna(seq))
   ...:
MHMLDENLYDLGMKACHEGTNVLDKWRNMARVCSCDYQFK
MQGSDGQQESYCTLPFEVSGMP
MPVEWRTMQFQRLERASCVKDSTFKNTGSFIKDRKVSGISQDEWAYAMSHQMQPAAHYA
MIVVTMCQ
MGQCMRFAPGMHGMYSSFHPQHKEITPGIDYASMNEVETAETIRPI
```

5.1.4 Exercises

Problems

P5.1.1 Improve on the algorithm to find the number of factors of an integer given in Section 5.1.3 by (a) looping the trial factor, `i`, up to no greater than the square root of `n` (why is it not necessary to test values of `i` greater than this?), and (b) using a generator (see Section 4.3.5). Compare the execution speed of these alternatives using the `%timeit` IPython magic.

P5.1.2 Using the fastest algorithm from the previous question, devise a short piece of code to determine the *highly composite numbers* less than 100000 and use the `%%timeit` cell magic to time its execution. A highly composite number is a positive integer with more factors than any smaller positive integer, for example: 1, 2, 4, 6, 12, 24, 36, 48, $\cdots$.

5.2 IPython Notebook

IPython Notebook provides an interactive environment for Python programming within a web browser. Its main advantage over the more traditional console-based approach of the IPython shell is that Python code can be combined with documentation (including in rendered LaTeX), images and even rich media such as embedded videos. IPython notebooks are increasingly being used by scientists to communicate their research by including the computations carried out on data as well as simply the results of those computations. The format makes it easy for researchers to collaborate on a project and for others to validate their findings by reproducing their calculations on the same data. Note that from version 4, the IPython Notebook project has been reformulated as Jupyter with bindings for other languages as well as Python.

5.2.1 IPython notebook basics

Starting the IPython notebook server

If you have IPython notebook installed, the server that runs the browser-based interface to IPython can be started from the command line with

```
ipython notebook
```

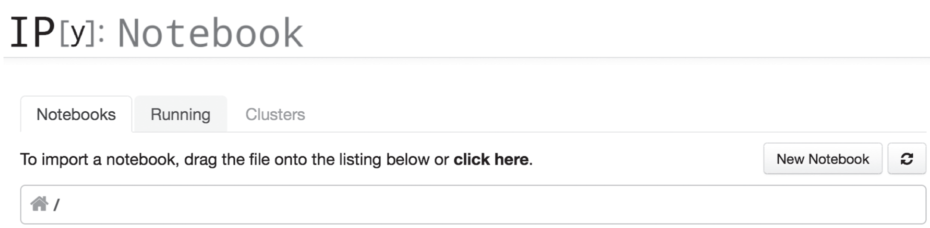


Figure 5.1 The IPython notebook index page.

This will open a web browser window at the URL of the local IPython notebook application. By default this is http://127.0.0.1:8888 though it will default to a different port if 8888 is in use.

The notebook index page (Figure 5.1) contains a list of the notebooks currently available in the directory from which the notebook server was started. This is also the default directory to which notebooks will be saved (with the extension `.ipynb`), so it is a good idea to execute the above command somewhere convenient in your directory hierarchy for the project you are working on.

The index page contains three tabs: *Notebooks* lists the IPython notebooks and subdirectories within the current working directory, *Running* lists those notebooks that are currently active within your session (even if they are not open in a browser window); *Clusters* provides an interface to IPython's parallel computing engine: we will not cover this topic in this book.

From the index page, one can start a new notebook (by clicking on "New Notebook") or open an existing notebook (by clicking on its name). To import an existing notebook into the index page, either click where indicated at the top of the page or drag the notebook file into the index listing from elsewhere on your operating system.

To stop the notebook server, press CTRL-C in the terminal window it was started from (and confirm at the prompt).

Editing an IPython notebook

To start a new notebook, click the "New Notebook" button. This opens a new browser tab containing the interface where you will write your code and connects it to an IPython *kernel*, the process responsible for executing the code and communicating the results back to the browser.

The new notebook document (Figure 5.2) consists of a *title bar*, a *menu bar* and a *tool bar*, under which is an IPython prompt where you will type the code and markup (e.g., explanatory text and documentation) as a series of *cells*.

In the title bar the name of the first notebook you open will probably be "Untitled0"; click on it to rename it to something more informative. The menu bar contains options for saving, copying, printing, rearranging and otherwise manipulating the notebook document. The tool bar consists of series of icons that act as shortcuts for common operations that can also be achieved through the menu bar.

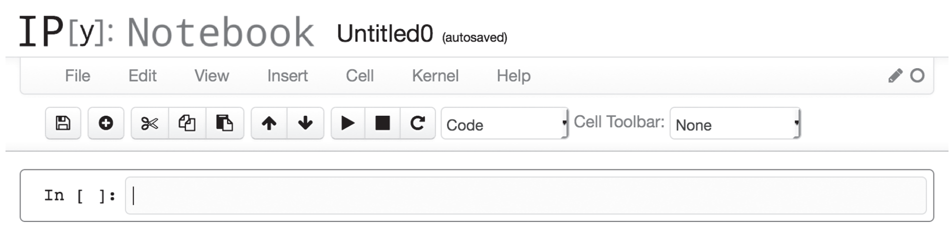


Figure 5.2 IPython with a new notebook document.

There are four types of input cells where you can write the content for your notebook:

- Code cells: the default type of cell, this type of cell consists of executable code. As far as this chapter is concerned, the code you write here will be Python, but IPython Notebook (now called Jupyter) does provide a mechanism of executing code written in other languages such as Julia and R.
- Heading cells: six levels of heading (from top-level section titles to paragraph-level text). When "executed" this type of cell produces a rich-text rendering of their contents at an appropriate font size.
- Markdown cells: this type of cell allows for a rich form of documentation for your code. When executed, the input to a markdown cell is converted into HTML, which can include mathematical equations, font effects, lists, tables, embedded images and videos – see Section 5.2.1.
- Raw cells: input into this type of cell is not changed by the notebook – its content and formatting is preserved exactly.

Running cells

Each cell can consist of more than one line of input, and the cell is not interpreted until you "run" (i.e., execute) it. This is achieved either by selecting the appropriate option from the menu bar (under the "Cell" drop-down submenu), by clicking the "Run cell" "play" button on the tool bar, or through the following keyboard shortcuts:

- `Shift-Enter`: Execute the cell, showing any output, and then *move the cursor* onto the cell below. If there is no cell below, a new, empty one will be created.
- `CTRL-Enter`: Execute the cell in place, but *keep the cursor* in the current cell. Useful for quick "disposable" commands to check if a command works or for retrieving a directory listing.
- `Alt-Enter`: Execute the cell, showing any output, and then *insert and move the cursor to a new cell* immediately beneath it.

Two other keyboard shortcuts are useful. When editing a cell the arrow keys navigate the *contents* of the cell (*edit mode*); from this mode, pressing `Esc` enters *command mode* from which the arrow keys navigate *through* the cells. To reenter edit mode on a selected cell, press `Enter`.

The menu bar, under the "Cell" drop-down submenu, provides many ways of running a notebook's cells: usually, you will want to run the current cell individually or run it and all those below it.

Code cells

You can enter anything into a code cell that you can when writing a Python program in an editor or at the regular IPython shell. Code in a given cell has access to objects defined in other cells (providing they have been run). For example,

```
In [ ]:  n = 10
```

Pressing Shift-Enter or clicking Run Cell executes this statement (defining `n` but producing no output) and opens a new cell underneath the old one:

```
In [1]:  n = 10

In [ ]:
```

Entering the following statements at this new prompt:

```
In [ ]:  sum_of_squares = n * (n+1) * (2*n+1) // 6
         print('1**2 + 2**2 + ... + {}**2 = {}'.format(n,
             sum_of_squares))
```

and executing as before produces output and opens a third empty input cell. The whole notebook document then looks like:

```
In [1]:  n = 10

In [2]:  sum_of_squares = n * (n+1) * (2*n+1) // 6
         print('1**2 + 2**2 + ... + {}**2 = {}'.format(n,
             sum_of_squares))

Out[2]:  1**2 + 2**2 + ... + 10**2 = 385

In [ ]:
```

You can edit the value of `n` in input cell 1 and rerun the entire document to update the output. It is worth noting that it is also possible to set a new value for `n` *after* the calculation in cell 2:

```
In [3]:  n = 15
```

running cell 3 and then cell 2 then leaves the output to cell 2 as

```
Out[2]:  1**2 + 2**2 + ... + 15**2 = 1240
```

even though the cell above still defines `n` to be 10. That is, unless you run the entire document from the beginning, the output does not necessarily reflect the output of a script corresponding to the code cells taken in order.

System commands (those prefixed with ! or !!) and IPython magics can all be used within IPython notebook.

It is also possible to use `pylab` "inline" in the notebook so that plots show up as images embedded in the document. To turn this feature on, use

```
In [x]:  %pylab inline
```

By itself this command imports the `pylab` library we used in Chapter 3 (and a few other things besides), but imports its symbols into the namespace of your interactive session. That is, the `%pylab inline` magic has the effect of `from pylab import *` and you can type, for example, `plot(x, y)` instead of `pylab.plot(x, y)`. To prevent this behavior, we recommend adding the argument `--no-import-all`:

```
In [x]:  %pylab inline --no-import-all
```

This stops `pylab` from polluting your namespace with its own definitions.[7]

Markdown cells

Markdown cells convert your input text into HTML, applying styles according to a simple syntax illustrated below. The full documentation is at

http://daringfireball.net/projects/markdown/

Here we explain the most useful features. A complete notebook of these examples can be downloaded from scipython.com/book/markdown.

Basic markdown

- Simple styles can be applied by enclosing text by asterisks or underscores:

```
In [x]:  Surrounding text by two asterisks denotes
             **bold style**; using one asterisk denotes
             *italic text*, as does _a single
             underscore_.
```

Surrounding text by two asterisks denotes **bold style**; using one asterisk denotes *italic text*, as does *a single underscore*.

- Block quotes are indicated by a single angle bracket, >:

```
In [x]:  > "Climb if you will, but remember that
             courage and strength are nought without
             prudence, and that a momentary negligence
             may destroy the happiness of a lifetime.
             Do nothing in haste; look well to each
             step; and from the beginning think what
             may be the end." - Edward Whymper
```

> “Climb if you will, but remember that courage and strength are nought without prudence, and that a momentary negligence may destroy the happiness of a lifetime. Do nothing in haste; look well to each step; and from the beginning think what may be the end.” – Edward Whymper

- Code *examples* (for illustration rather than execution) are between blank lines and indented by four spaces (or a tab). The following will appear in a monospaced font with the characters as entered:

[7] It is particularly annoying to find your innocent variable names such as `f` clash with `pylab`’s own function calls.

In [x]:
```
    n = 57
    while n != 1:
        if n % 2:
            n = 3*n + 1
        else:
            n //= 2
```

```
n = 57
while n != 1:
    if n % 2:
        n = 3*n + 1
    else:
        n //= 2
```

- Inline code examples are created by surrounding the text with backticks (`):

In [x]:
```
Here are some Python keywords: `for`, `while`
    and `lambda`.
```

Here are some Python keywords: `for`, `while` and `lambda`.

- New paragraphs are started after a blank line.

HTML within markdown

The markdown used by IPython notebooks encompasses HTML, so valid HTML entities and tags can be used directly: for example, the `<em>` tag for emphasis, as can CSS styles to produce effects such as underlined text. Even complex HTML such as tables can be marked up directly.

In [x]:
```
The following <em>Punnett table</em> is <span
    style="text-decoration: underline" >marked
    up</span> in HTML.

<table style="text-align: center;">
<tr>
<th style="border-top:none; border-left:none;"
    rowspan="2" colspan="2"></th>
<th colspan="2">Male</th>
</tr>
<tr>
<th>A</th>
<th>a</th>
</tr>
<tr>
<th rowspan="2">Female</th>
<th>a</th>
<td style="background: #aaa;">Aa</td>
<td>aa</td>
</tr>
<tr>
<th>a</th>
<td style="background: #aaa;">Aa</td>
<td>aa</td>
</tr>
</table>
```

The following *Punnett table* is marked up in HTML.

<table>
<tr><td colspan="2" rowspan="2"></td><th colspan="2">Male</th></tr>
<tr><th>A</th><th>a</th></tr>
<tr><th rowspan="2">Female</th><th>a</th><td>Aa</td><td>aa</td></tr>
<tr><th>a</th><td>Aa</td><td>aa</td></tr>
</table>

Lists

Itemized (unnumbered) lists are created using any of the markers `*`, `+` or `-`, and nested sublists are simply indented.

In [x]:
```
The inner planets and their satellites:

* Mercury
* Venus
* Earth
    * The Moon
+ Mars
    - Phoebus
    - Deimos
```

The inner planets and their satellites:

- Mercury
- Venus
- Earth
  - The Moon
- Mars
  - Phoebus
  - Deimos

Ordered (numbered) lists are created by preceding items by a number followed by a full stop (period) and a space:

In [x]:
```
1. Symphony No. 1 in C major, Op. 21
2. Symphony No. 2 in D major, Op. 36
3. Symphony No. 3 in E-flat major ("Eroica"), Op. 55
```

1. Symphony No. 1 in C major, Op. 21
2. Symphony No. 2 in D major, Op. 36
3. Symphony No. 3 in E-flat major ("Eroica"), Op. 55

Links

There are three ways of introducing links into markdown text:

- *Inline* links provide a URL in round brackets after the text to be turned into a link in square brackets. For example,

In [x]:
```
Here is a link to the
[IPython website](http://ipython.org/).
```

Here is a link to the IPython website.

- *Reference* links label the text to turn into a link by placing a name (containing letters, numbers or spaces) in square brackets after it. This name is expected to be defined using the syntax `[name]: url` elsewhere in the document, as in the following example markdown cell.

In [x]:
```
Some important mathematical sequences are the
    [prime numbers][primes],
[Fibonacci sequence][fib] and the [Catalan
    numbers][catalan_numbers].

...

[primes]: http://oeis.org/A000040
[fib]: http://oeis.org/A000045
[catalan_numbers]: http://oeis.org/A000108]
```

Some important mathematical sequences are the primes, Fibonacci sequence and the Catalan numbers.

- *Automatic* links, for which the clickable text is the same as the URL are created simply by surrounding the URL by angle brackets:

In [x]:
```
My website is <http://www.christianhill.co.uk>.
```

My website is http://www.christianhill.co.uk.

If the link is to a file on your local system, give as the URL the path, relative to the notebook directory, prefixed with `files/`:

In [x]:
```
Here is [a local data file](files/data/data0.txt).
```

Here is a a local data file.

Note that links open in a new browser tab when clicked.

Mathematics

Mathematical equations can be written in LaTeX and are rendered using the Javascript library, MathJax. Inline equations are delimited by single dollar signs; “displayed” equations by doubled dollar signs:

In [x]:
```
An inline equation appears within a sentence of
    text, as in the definition of the function
    $f(x) = \sin(x^2)$; displayed equations get
    their own line(s) between lines of text:
$$\int_0^\infty e^{-x^2}dx = \frac{\sqrt{\pi}}{2}.$$
```

An inline equation appears within a sentence of text, as in the definition of the function $f(x) = \sin(x^2)$; displayed equations get their own line(s) between lines of text:

$$\int_0^\infty e^{-x^2} dx = \frac{\sqrt{\pi}}{2}.$$

Images and video

Links to image files work in exactly the same way as ordinary links (and can be inline or reference links), but are preceded by an exclamation mark, `!`. The text in square brackets between the exclamation mark and the link acts as *alt text* to the image. For example,

```
In [x]:  ![An interesting plot of the Newton
             fractal](/files/images/newton_fractal.png)
         ![A remote link to a star
             image](http://christianhill.co.uk/media/books/
             python/star.svg)
```

Video links must use the HTML5 `<video>` tag, but note that not all browsers support all video formats. For example,

```
In [x]:  <video controls style="width: 500px; margin: 0
             auto; display: block;"
             src="files/diffmap-animated.ogv" />
```

The data constituting images, video and other locally linked content are not *embedded* in the notebook document itself: these files must be provided with the notebook when it is distributed.

5.2.2 Converting notebooks to other formats

`nbconvert` is a tool, installed with IPython notebook, to convert notebooks from their native `.ipynb` format[8] to any of several alternative formats. It is run from the (system) command line as

```
ipython nbconvert --to <format> <notebook.ipynb>
```

where *`notebook.ipynb`* is the name of the IPython notebook file to be converted and *`format`* is the desired output format. The default (if no *`format`* is given), is to produce a static HTML file, as described below.

Conversion to HTML

The command

```
ipython nbconvert <notebook.ipynb>
```

converts `notebook.ipynb` to HTML and produces a file, `notebook.html` in the current directory. This file contains all the necessary headers for a stand-alone HTML page,

[8] This format is, in fact, just a JSON (JavaScript Object Notation) document.

which will closely resemble the interactive view produced by the IPython notebook server, but as a static document.

If you want just the HTML corresponding to the notebook without the header (`<html>`, `<head>`, `<body>` tags, etc.), suitable for embedding in an existing web page, add the `--template basic` option.

Any supporting files, such as images, are automatically placed in a directory with the same base name as the notebook but suffixes with `_files`. For example, `ipython nbconvert mynotebook.ipynb` generates `mynotebook.html` and the directory `mynotebook_files`.

Conversion to LaTeX

To export the notebook as a LaTeX document, use

```
ipython nbconvert --to latex <notebook.ipynb>
```

To automatically run `pdflatex` on the `notebook.tex` file generated to produce a PDF file, add the option --`post pdf`.

Conversion to markdown

```
ipython nbconvert --to markdown <notebook.ipynb>
```

converts the whole notebook into markdown (see Section 5.2.1): cells that are already in markdown are unaffected and code cells are placed in triple-backtick (` ``` `) blocks.

Conversion to Python

The command

```
ipython nbconvert --to python <notebook.ipynb>
```

converts `notebook.ipynb` into an executable Python script. If any of the notebook's code cells contain IPython magic functions, this script may only be executable from within an IPython session. Markdown and other text cells are converted to comments in the generated Python script code.

6 NumPy

NumPy has become the de facto standard package for general scientific programming in Python. Its core object is the `ndarray`, a multidimensional array of a single data type which can be sorted, reshaped, subject to mathematical operations and statistical analysis, written to and read from files, and much more. The NumPy implementations of these mathematical operations and algorithms have two main advantages over the "core" Python objects we have used until now. First, they are implemented as precompiled C code and so approach the speed of execution of a program written in C itself; second, NumPy supports *vectorization*: a single operation can be carried out on an entire array, rather than requiring an explicit loop over the array's elements. For example, compare the multiplication of two one-dimensional lists of `n` numbers, `a` and `b`, in the core python language:

```
c = []
for i in range(n):
    c.append(a[i] * b[i])
```

and using NumPy arrays:[1]

```
c = a * b
```

The elementwise multiplication is handled by optimized, precompiled C and so is very fast (much faster for large `n` than the core Python alternative). The absence of explicit looping and indexing makes the code cleaner, less error-prone and closer to the standard mathematical notation it reflects.

All of NumPy's functionality is provided by the `numpy` package. To use it, it is strongly advised to import with

```
import numpy as np
```

and then to refer to its attributes with the prefix `np.` (e.g., `np.array`). This is the way we use NumPy in this book.

6.1 Basic array methods

The NumPy array class is `ndarray`, which consists of a multidimensional table of elements indexed by a tuple of integers. Unlike Python lists and tuples, the elements

[1] We will use the terms NumPy array and `ndarray` interchangeably

cannot be of different types: each element in a NumPy array has the same type, which is specified by an associated *data type* object (`dtype`). The `dtype` of an array specifies not only the broad class of element (integer, floating point number, etc.) but also how it is represented in memory (e.g., how many bits it occupies) – see Section 6.1.2.

The dimensions of a NumPy array are called *axes*; the number of axes an array has is called its *rank*.

6.1.1 Creating an array

Basic array creation

The simplest way to create a small NumPy array is to call the `np.array` constructor with a list or tuple of values:

```
In [x]: import numpy as np
In [x]: a = np.array( (100, 101, 102, 103) )
In [x]: a
Out[x]: array([100, 101, 102, 103])
In [x]: b = np.array( [[1.,2.], [3.,4.]] )
Out[x]:
array([[ 1.,  2.],
       [ 3.,  4.]])
```

Note that passing a list of lists creates a two-dimensional array (and similarly for higher dimensions).

Indexing a multidimensional NumPy array is a little different from indexing a conventional Python list of lists: instead of `b[i][j]`, refer to the index of the required element as a tuple of integers, `b[i,j]`:

```
In [x]: b[0,1]          # same as b[(0,1)]
Out[x]: 2.0
In [x]: b[1,1] = 0.     # also for assignment
Out[x]:
array([[ 1.,  2.],
       [ 3.,  0.]])
```

The data type is deduced from the type of the elements in the sequence and "upcast" to the most general type if they are of mixed but compatible types:

```
In [x]: np.array( [-1, 0, 2.])  # mixture of int and float: upcast to float
Out[x]: array([-1.,  0.,  2.])
```

You can also explicitly set the data type using the optional `dtype` argument (see Section 6.1.2):

```
In [x]: np.array( [0, 4, -4], dtype=complex)
In [x]: array([ 0.+0.j,  4.+0.j, -4.+0.j])
```

If your array is large or you do not know the element values at the time of creation, there are several methods to declare an array of a particular shape filled with default or arbitrary values. The simplest and fastest, `np.empty`, takes a tuple of the array's shape and creates the array without initializing its elements: the initial element values

are undefined (typically random junk defined from whatever were the contents of the memory that Python allocated for the array).

```
In [x]: np.empty((2,2))
Out[x]:
array([[ -2.31584178e+077,  -1.72723381e-077],
       [  2.15686807e-314,   2.78134366e-309]])
```

There are also helper methods `np.zeros` and `np.ones`, which create an array of the specified shape with elements prefilled with 0 and 1 respectively. `np.empty`, `np.zeros` and `np.ones` also take the optional `dtype` argument.

```
In [x]: np.zeros((3,2))     # default dtype is 'float'
Out[x]:
array([[ 0.,  0.],
       [ 0.,  0.],
       [ 0.,  0.]])
In [x]: np.ones((3,3), dtype=int)
Out[x]:
array([[1, 1, 1],
       [1, 1, 1],
       [1, 1, 1]])
```

If you already have an array and would like to create another with the same shape, `np.empty_like`, `np.zeros_like` and `np.ones_like` will do that for you:

```
In [x]: a
Out[x]: array([100, 101, 102, 103])
In [x]: np.ones_like(a)
Out[x]: array([1, 1, 1, 1])
In [x]: np.zeros_like(a, dtype=float)
Out[x]: array([ 0.,  0.,  0.,  0.])
```

Note that the array created inherits its `dtype` from the original array; to set its data type to something else, use the `dtype` argument.

Initializing an array from a sequence

To create an array containing a sequence of numbers there are two methods: `np.arange` and `np.linspace`. `np.arange` is the NumPy equivalent of `range`, except that it can generate floating point sequences. It also actually allocates the memory for the elements in an `ndarray` instead of returning a generator-like object – compare Section 2.4.3.

```
In [x]: np.arange(7)
Out[x]: array([0, 1, 2, 3, 4, 5, 6])
In [x]: np.arange(1.5, 3., 0.5)
Out[x]: array([ 1.5,  2. ,  2.5]))
```

As with `range` the array generated in these examples does not include the last elements, 7 and 3. However, `arange` has a problem: because of the finite precision of floating point arithmetic it is not always possible to know how many elements will be created. For this reason, and because one often often wants the last element of a specifed sequence, the `np.linspace` function can be a more useful way of creating

an sequence.[2] For example, to generate an evenly spaced array of the five numbers between 1 and 20 *inclusive*:

```
In [x]: np.linspace(1, 20, 5)
Out[x]: array([  1.  ,   5.75,  10.5 ,  15.25,  20.  ])
```

`np.linspace` has a couple of optional boolean arguments. First, setting `retstep` to `True` returns the number spacing (step size):

```
In [x]: x, dx = np.linspace(0., 2*np.pi, 100, retstep=True)
In [x]: dx
Out[x]: 0.06346651825433926
```

This saves you from calculating `dx = (end-start)/(num-1)` separately; in this example, the 100 points between 0 and 2π inclusive are spaced by $2\pi/99 = 0.0634665\cdots$. Finally, setting `endpoint` to `False` omits the final point in the sequence, as for `np.arange`:

```
In [x]: x = np.linspace(0, 5, 5, endpoint=False)
Out[x]: array([ 0.,  1.,  2.,  3.,  4.])
```

Note that the array generated by `np.linspace` has the `dtype` of floating point numbers, even if the sequence generates integers.

Initializing an array from a function

To create an array initialized with values calculated using a function, use NumPy's `np.fromfunction` method, which takes as its arguments a function and a tuple representing the shape of the desired array. The function should itself take the same number of arguments as dimensions in the array: these arguments index each element at which the function returns a value. An example will make this clearer:

```
In [x]: def f(i, j):
   ...:     return 2 * i * j
   ...:
In [x]: np.fromfunction(f,(4,3))
array([[  0.,   0.,   0.],
       [  0.,   2.,   4.],
       [  0.,   4.,   8.],
       [  0.,   6.,  12.]])
```

The function `f` is called for every index in the specified shape and the values it returns are used to initialize the corresponding elements.[3] A simple expression like this one can be replaced by an anonymous `lambda` function (see Section 4.3.3) if desired:

```
In [x]: np.fromfunction(lambda i,j: 2*i*j, (4,3))
```

Example E6.1 To create a "comb" of values in an array of length N for which every nth element is one but with zeros everywhere else:

```
In [x]: N, n = 101, 5
In [x]: def f(i):
```

[2] We came across `linspace` in Example E3.1.

[3] Note that the indexes are passed as `ndarrays` and expect the function, `f`, to use vectorized operations.

```
   ...:     return (i % n == 0) * 1
   ...:
In [x]: comb = np.fromfunction(f, (N,), dtype=int)
In [x]: print(comb)
[1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1
 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1 0
 0 0 0 1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1]
```

`ndarray` attributes for introspection

A NumPy array knows its rank, shape, size, `dtype` and one or two other properties: these can be determined directly from the attributes described in Table 6.1. For example,

```
In [x]: a = np.array(((1, 0, 1), (0, 1, 0)))
In [x]: a.shape
Out[x]: (2, 3)      # 2 rows, 3 columns
In [x]: a.ndim      # rank (number of dimensions)
Out[x]: 2
In [x]: a.size      # total number of elements
Out[x]: 6
In [x]: a.dtype
Out[x]: dtype('int64')
In [x]: a.data
Out[x]: <memory at 0x102387308>
```

The `shape` attribute returns the axis dimensions in the same order as the axes are indexed: a two-dimensional array with `n` rows and `m` columns has a `shape` of `(n,m)`.

6.1.2 NumPy's basic data types (`dtype`s)

So far, the NumPy arrays we have created have contained either integers or floating point numbers, and we have let Python take care of the details of how these are represented. However, NumPy provides a powerful way of determining these details explicitly using *data type* objects. This is necessary, because in order to interface with the underlying compiled C code the elements of a NumPy array must be stored in a compatible

Table 6.1 `ndarray` Attributes

Attribute	Description
`shape`	The array dimensions: the size of the array along each of its axes, returned as a tuple of integers
`ndim`	Number of axes (dimensions). Note that `ndim == len(shape)`
`size`	The total number of elements in the array, equal to the product of the elements of `shape`
`dtype`	The array's data type (see Section 6.1.2)
`data`	The "buffer" in memory containing the actual elements of the array
`itemsize`	The size in bytes of each element

Table 6.2 Commmon NumPy data types

Data Type	Description
`int_`	The default integer type, corresponding to C's `long`: *platform-dependent*
`int8`	Integer in a single byte: -128 to 127
`int16`	Integer in 2 bytes: -32768 to 32767
`int32`	Integer in 4 bytes: -2147483648 to 2147483647
`int64`	Integer in 8 bytes: -2^{63} to $2^{63}-1$
`uint8`	Unsigned integer in a single byte: 0 to 255
`uint16`	Unsigned integer in 2 bytes: 0 to 65535
`uint32`	Unsigned integer in 4 bytes: 0 to 4294967295
`uint64`	Unsigned integer in 8 bytes: 0 to $2^{64}-1$
`float_`	The default floatng point number type, another name for `float64`
`float32`	Single-precision, signed float: $\sim 10^{-38}$ to $\sim 10^{38}$ with ~ 7 decimal digits of precision
`float64`	Double-precision, signed float: $\sim 10^{-308}$ to $\sim 10^{308}$ with ~ 15 decimal digits of precision
`complex_`	The default complex number type, another name for `complex128`
`complex64`	Single-precision complex number (represented by 32-bit floating point real and imaginary components)
`complex128`	Double-precision complex number (represented by 64-bit floating point real and imaginary components)
`bool_`	The default boolean type represented by a single byte

format: that is, each element is represented in a fixed number of bytes that are interpreted in a particular way.

For example, consider an *unsigned* integer stored in 2 bytes (16 bits) of memory (the C-type `uint16_t`). Such a number can take a value between 0 and $2^{16} - 1 = 65535$. No equivalent native Python type exists for this exact representation: Python integers are signed quantities and memory is dynamically assigned for them as required by their size. So NumPy defines a data type object, `np.uint16` to describe data stored in this way.

Furthermore, different systems can order the two bytes of this number differently, a distinction known as *endianness*. The *big-endian* convention places the most-significant byte in the smallest memory address; the *little-endian* convention places the least-significant byte in the smallest memory address. In creating your own arrays, NumPy will use the default convention for the hardware your program is running on, but it is essential to set the endianness correctly if reading in a binary file generated by a different computer.

Table 6.3 Common NumPy data type strings

String	Description
`i`	Signed integer
`u`	Unsigned integer
`f`	Floating point number[a]
`c`	Complex floating point number
`b`	Boolean value
`S, a`	String (fixed-length sequence of characters)
`U`	Unicode

[a] Note that without specifying the byte size, setting `dtype='f'` creates a *single-precision* floating point data type, equivalent to `np.float32`.

A full list of the numerical data types[4] is given in the NumPy documentation,[5] but the more common ones are listed in Table 6.2. They all exist within the `numpy` package and so can be referred to as, for example, `np.uint16`. The data types that get created by default when using the native Python numerical types are those with a trailing underscore: `np.float_`, `np.complex_` and `np.bool_`.

Apparently higher-precision floating point number data types such as `float96`, `float128` and `longdouble` are available but are not to be trusted: their implementation is platform dependent, and on many systems they do not actually offer any extra precision but simply align array elements on the appropriate byte-boundaries in memory.

To create a NumPy array of values using a particular data type, use the `dtype` argument of any array constructor function (such as `np.array`, `np.zeros`, etc.). This argument takes either a data type object (such as `np.uint8`) or something that can be converted into one. It is common to specify the `dtype` using a string consisting of a letter indicating the broad category of data type (integer, unsigned integer, complex number, etc.) optionally followed by a number giving the byte size of the type. For example,

```
In [x]: b = np.zeros((3,3), dtype='u4')
```

creates a 3×3 array of unsigned, 32-bit (4-byte) integers (equivalent to `np.uint32`). A list of supported data type letters and their meanings is given in Table 6.3.

To specify the endianness, use the prefixes `>` (big-endian), `<` (little-endian) or `|` (endianness not relevant). For example,

```
In [x]: a = np.zeros((3,3), dtype='>f8')
In [x]: b = np.zeros((3,3), dtype='<f')
In [x]: c = np.empty((3,3), dtype='|S4')
```

create arrays of big-endian double-precision numbers, little-endian single-precision numbers and four-character strings respecively.

[4] Strictly speaking, these types are *array scalar types* and not `dtypes`, but for our use here the distinction is not important.

[5] http://docs.scipy.org/doc/numpy/user/basics.types.html.

In these examples we have passed a *typecode* string to an array constructor's `dtype` argument, but it is also possible to create a `dtype` object first and pass that instead:

```
In [x]: dt = np.dtype('f8')
In [x]: dt
dtype('float64')    # i.e. 8 bytes, double-precision floating point
In [x]: a = np.array([0., 1., -2.], dtype=dt)
```

`dtype` objects have a handful of useful introspection methods:

```
In [x]: dt.str      # a string identifying the data type
'<f8'
In [x]: dt.name     # data type name and bit-width
'float64'
In [x]: dt.itemsize # data type size in bytes
8
```

To copy an array to a new array with a different data type, pass the desired `dtype` or typecode to the `astype` method:

```
In [x]: a = np.array([1.2345678, 2.5, 3.9])
In [x]: a.astype('float32')     # cast to single-precision float
Out[x]: array([ 1.23456776,  2.5       ,  3.9000001 ], dtype=float32)
In [x]: a.astype(np.uint8)      # cast to unsigned, 1-byte integer
Out[x]: array([1, 2, 3], dtype=uint8)
```

Strings in NumPy arrays are *byte strings* of a fixed size: each "character" is represented by a single byte, in contrast to the variable size UTF-8 encoding commonly used to represent Unicode strings. This is necessary because NumPy arrays have a predefined, fixed size in which all the elements occupy the same amount of memory so that they can be indexed efficiently with a constant stride. Unicode strings encoded with UTF-8, however, represent characters as code points with a variable width (see Section 2.3.3). Of course, any string is ultimately stored as a sequence of bytes and Python provides methods for translating between encodings. For example, on a system encoding strings with UTF-8 by default:

```
In [x]: s = 'piñata'        # UTF-8 encoded Unicode string
In [x]: b = s.encode()
In [x]: b
b'pi\xc3\xb1ata'            # byte string: ñ is stored in two bytes: hex C3B1
In [x]: len(s), len(b)
(6,7)                       # 6 UTF-8 encoded characters stored in 7 bytes
In [x]: arr = np.empty((2,2), 'S7')
In [x]: arr[:] = b          # Store the byte string b in array arr
In [x]:
array([[b'pi\xc3\xb1ata', b'pi\xc3\xb1ata'],
       [b'pi\xc3\xb1ata', b'pi\xc3\xb1ata']],
      dtype='|S7')
In [x]: arr[0,0]            # returns the byte string
b'pi\xc3\xb1ata'
In [x]: arr[0,0].decode()   # decode the byte string back assuming UTF-8
'piñata'
```

6.1.3 Universal functions (ufuncs)

In addition to the basic arithmetic operations of addition, division and more, NumPy provides many of the familiar mathematical functions that the math module (Section 2.2.2) does, implemented as so-called *universal functions* that act on each element of an array, producing an array in return without the need for an explicit loop. Universal functions are the way NumPy allows for *vectorization*, which promotes clean, efficient and easy-to-maintain code. For example,

```
In [x]: x = np.linspace(1,5,5)
In [x]: x**2
Out[x]: array([  1.,   4.,   9.,  16.,  25.])
In [x]: x - 1
Out[x]: array([ 0.,  1.,  2.,  3.,  4.])
In [x]: np.sqrt(x - 1)
Out[x]: array([ 0.,  1.,  1.41421356,  1.73205081,  2.])
In [x]: y = np.exp(-np.linspace(0., 2., 5))
In [x]: np.sin(x - y)
Out[x]: array([ 0.,  0.98431873,  0.48771645, -0.59340065, -0.98842844])
```

Array multiplication occurs *elementwise*: matrix multiplication is implemented by NumPy's dot function (or using matrix objects, see Section 6.6):

```
In [x]: a = np.array( ((1,2), (3,4)) )
In [x]: b = a
In [x]: a * b       # elementwise multiplication
Out[x]:
array([[ 1,  4],
       [ 9, 16]])
In [x]: a.dot(b)    # or np.dot(a, b)
Out[x]:
array([[ 7, 10],
       [15, 22]])
```

Comparison and logic operators (~, & and | for *not*, *and* and *or* respectively) are also vectorized and result in arrays of boolean values:

```
In [x]: a = np.linspace(1,6,6)**3
In [x]: print(a)
[   1.    8.   27.   64.  125.  216.]
In [x]: print(a > 100)
[False False False False  True  True]
In [x]: print((a < 10) | (a > 100))
[ True  True False False  True  True]
```

6.1.4 NumPy's special values, nan and inf

NumPy defines two special values to represent the outcome of calculations, which are not mathematically defined or not finite. The value np.nan ("not a number," NaN) represents the outcome of a calculation that is not a well-defined mathematical operation (e.g., 0/0); np.inf represents infinity.[6] For example,

[6] These quantities are defined in accordance with the IEEE 754 standard for floating point numbers.

```
In [x]: a = np.arange(4)
In [x]: a /= 0      # [0/0 1/0 2/0 3/0]
In [x]: a
Out[x]: array([ nan,  inf,  inf,  inf])
```

Do not test nans for equality (`np.nan == np.nan` is `False`). Instead, NumPy provides methods `np.isnan`, `np.isinf` and `np.isfinite`:

```
In [x]: np.isnan(a)
Out[x]: array([ True, False, False, False], dtype=bool)
In [x]: np.isinf(a)
Out[x]: array([False,  True,  True,  True], dtype=bool)
In [x]: np.isfinite(a)
Out[x]: array([False, False, False, False], dtype=bool)
```

Note that `nan` is neither finite nor infinite! (See also Section 9.1.4.)

Example E6.2 A *magic square* is an $N \times N$ grid of numbers in which the entries in each row, column and main diagonal sum to the same number (equal to $N(N^2 + 1)/2$). A method for constructing a magic square for odd N is as follows:

Step 1. Start in the middle of the top row, and let $n = 1$;

Step 2. Insert n into the current grid position;

Step 3. If $n = N^2$ the grid is complete so stop. Otherwise, increment n;

Step 4. Move diagonally up and right, wrapping to the first column or last row if the move leads outside the grid. If this cell is already filled, move vertically down one space instead;

Step 5. Return to step 2.

The following program creates and displays a magic square.

Listing 6.1 Creating a magic square

```
# Create an N x N magic square. N must be odd.
import numpy as np

N  = 5
magic_square = np.zeros((N,N), dtype=int)

n = 1
i, j = 0, N//2

while n <= N**2:
    magic_square[i, j] = n
    n += 1
    newi, newj = (i-1) % N, (j+1)% N
    if magic_square[newi, newj]:
        i += 1
    else:
        i, j = newi, newj

print(magic_square)
```

The 5 × 5 magic square output by the earlier example is

```
[[17 24  1  8 15]
 [23  5  7 14 16]
 [ 4  6 13 20 22]
 [10 12 19 21  3]
 [11 18 25  2  9]]
```

6.1.5 Changing the shape of an array

Whatever the rank of an array, its elements are stored in sequential memory locations that are addressed by a single index (internally, the array is one-dimensional, but knowing the shape of the array, Python is able to resolve a tuple of indexes into a single memory address). NumPy's arrays are stored in memory in C-style, *row-major* order, that is, with the elements of the last (rightmost) index stored contiguously. In a two-dimensional array, for example, the element `a[0,0]` is followed by `a[0,1]`. The array that follows

```
In [x]: a = np.array( ((1,2),(3,4)) )
In [x}: print(a)
[[1 2]
 [3 4]]
```

is stored in memory as the sequential elements `[1,2,3,4]`.[7]

`flatten` and `ravel`

Suppose you wish to "flatten" a multidimensional array onto a single axis. NumPy provides two methods to do this: `flatten` and `ravel`. Both flatten the array into its internal (row-major) ordering, as described earlier. `flatten` returns an independent *copy* of the elements and is generally slower than `ravel` which, tries to return a *view* to the flattened array. An array view is a new NumPy array with, in this case, a different shape from the original, but it does not "own" its data elements: it references the elements of another array. Thus, just as with mutable lists (Section 2.4.1), a reassignment of an element of one array affects the other. An example should make this clear:

```
In [x]: a = np.array( [[1,2,3], [4,5,6], [7,8,9]] )
In [x]: b = a.flatten()     # create and independent, flattened copy of a
In [x]: b
Out[x]: array([1, 2, 3, 4, 5, 6, 7, 8, 9])
In [x]: b[3] = 0
In [x]: b
Out[x]: array([1, 2, 3, 0, 5, 6, 7, 8, 9])
In [x]: a       # a is unchanged
Out[x]:
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
```

[7] This contrasts with Fortran's *column-major* ordering, which would store the elements as `[1,3,2,4]`.

Assignment to `b` didn't change `a` because they are completely independent objects that do not share their data. In contrast, the flattened array created by taking a view on `a` with `ravel` refers to the same underlying data:

```
In [x]: c = a.ravel()
In [x]: c
Out[x]:array([1, 2, 3, 4, 5, 6, 7, 8, 9])
In [x]: c[3] = 0
In [x]: c
Out[x]: array([1, 2, 3, 0, 5, 6, 7, 8, 9])
In [x]: a
Out[x]:
array([[1, 2, 3],
       [0, 5, 6],
       [7, 8, 9]])
```

You should be aware that although the `ravel` method "does its best" to return a view to the underlying data, various array operations (including *slicing*; see Section 6.1.6) can leave the elements stored in noncontiguous memory locations in which case `ravel` has no choice but to make a copy.

`resize` and `reshape`

An array may be resized (in place) to a compatible shape[8] with the `resize` method, which takes the new dimensions as its arguments. If the array doesn't reference another array's data and doesn't have references to it, resizing to a smaller shape is allowed and truncates the array; resizing to a larger shape pads with zeros. Array references are created when, for example, one array is a view on another (they share data) or simply by assignment: (`b=a`).

```
In [x]: a = np.linspace(1, 4, 4)    # the array [1. 2. 3. 4.]
Out[x]: print(a)
[1. 2. 3. 4.]
In [x]: a.resize(2,2)   # reshapes a in place, doesn't return anything
In [x]: print(a)
[[ 1.  2.]
 [ 3.  4.]]
In [x]: a.resize(3,2)   # OK: nothing else references a
In [x]: print(a)
[[ 1.  2.]
 [ 3.  4.]
 [ 0.  0.]]
```

The `reshape` method returns a view on the array with its elements reshaped as required. The original array is not modified.

```
In [x]: a = np.linspace(1, 4, 4)
In [x]: a.resize(3,2)
In [x]: a
[[ 1.  2.]
 [ 3.  4.]
 [ 0.  0.]]
```

[8] That is, a shape with the same total number of elements.

```
In [x]: b = a.reshape(6)
In [x]: print(b)
[ 1.  2.  3.  4.  0.  0.]
In [x]: b.resize(3,2)   # OK: same number of elements
In [x]: b.resize(2,2)   # not OK: b is a view on (shares) the same data as a
...

ValueError: cannot resize this array: it does not own its data
In [x]: a.resize(2,2)   # also not OK: a shares its data with b
ValueError: cannot resize this array: it does not own its data
```

Transposing an array

The method `transpose` returns a view of an array with the axes transposed. For a two-dimensional array, this is the usual matrix transpose:

```
In [x]: a = np.linspace(1,6,6).reshape(3,2)
In [x]: a
Out[x]:
array([[ 1.,  2.],
       [ 3.,  4.],
       [ 5.,  6.]])
In [x]: a.transpose()          # or simply a.T
Out[x]:
array([[ 1.,  3.,  5.],
       [ 2.,  4.,  6.]]
```

Note that transposing a one-dimensional array returns the array unchanged:

```
In [x]: b = np.array([100, 101, 102, 103])
In [x]: b.transpose()
Out[x]: array([100, 101, 102, 103])
```

The `np.matrix` object has methods for converting between column and row vectors if this is what you want; see also Section 6.1.6.

Merging and splitting arrays

A clutch of NumPy methods merge and split arrays in different ways. `np.vstack`, `np.hstack` and `np.dstack` stack arrays vertically (in sequential rows), horizontally (in sequential columns) and depthwise (along a third axis). For example,

```
In [x]: a = np.array([0, 0, 0, 0])
In [x]: b = np.array([1, 1, 1, 1])
In [x]: c = np.array([2, 2, 2, 2])
In [x]: np.vstack((a,b,c))
Out[x]:
array([[0, 0, 0, 0],
       [1, 1, 1, 1],
       [2, 2, 2, 2]])
In [x]: np.hstack((a,b,c))
Out[x]:
array([0, 0, 0, 0, 1, 1, 1, 1, 2, 2, 2, 2])
In [x]: np.dstack((a,b,c))
Out[x]:
array([[[0, 1, 2],
```

```
        [0, 1, 2],
        [0, 1, 2],
        [0, 1, 2]]])
```

Note that the array created contains an independent *copy* of the data from the original arrays.[9]

The inverse operations, `np.vsplit`, `np.hsplit` and `np.dsplit` split a single array into multiple arrays by rows, columns or depth. In addition to the array to be split, these methods require an argument indicating how to split the array. If this argument is a *single integer*, the array is split into that number of equal-sized arrays along the appropriate axis. For example,

```
In [x]: a = np.arange(6)
In [x]: a
Out[x]: array([ 0,  1,  2,  3,  4,  5])
In [x]: np.hsplit(a, 3)
Out[x]: [array([ 0,  1]), array([ 2,  3]), array([ 4,  5])]
```

– a list of array objects is returned. If the second argument is a sequence of integer indexes, the array is split on those indexes:

```
In [x]: a
Out[x]: array([ 0,  1,  2,  3,  4,  5])
In [x]: np.hsplit(a, (2, 3, 5))
[array([0, 1]), array([2]), array([3, 4]), array([5])]
```

– this is the same as the list `[a[:2], a[2:3], a[3:5], a[5:]]`. Unlike with `np.hstack`, etc., the arrays returned are *views* on the original data.[10]

Example E6.3 Suppose you have a 3×3 array to which you wish to add a row or column. Adding a row is easy with `np.vstack`:

```
In [x]: a = np.ones((3, 3))
In [x]: np.vstack( (a, np.array((2,2,2))) )
Out[x]:
array([[ 1.,  1.,  1.],
       [ 1.,  1.,  1.],
       [ 1.,  1.,  1.],
       [ 2.,  2.,  2.]])
```

Adding a column requires a bit more work, however. You can't use `np.hstack` directly:

```
In [x]: a = np.ones((3, 3))
In [x]: np.hstack( (a, np.array((2,2,2))) )

... [Traceback information] ...
ValueError: all the input arrays must have same number of dimensions
```

---

[9] NumPy has to copy the data because it has to store its data in one contiguous block of memory and the original arrays may be dispersed in different noncontiguous locations.

[10] NumPy does this for efficiency reasons – copying large amounts of data is expensive and not necessary to fulfill the function of these splitting methods.

This is because `np.hstack` cannot concatenate two arrays with different numbers of rows. Schematically:

```
[[ 1.,  1.,  1.],       [2., 2., 2.]
 [ 1.,  1.,  1.],   +                  = ?
 [ 1.,  1.,  1.]]
```

We can't simply transpose our new row, either, because it's a one-dimensional array and its transpose is the same shape as the original. So we need to *reshape* it first:

```
In [x]: a = np.ones((3, 3))
In [x]: b = np.array((2,2,2)).reshape(3,1)
In [x]: b
array([[2],
       [2],
       [2]])
In [x]: np.hstack((a, b))
Out[x]:
array([[ 1.,  1.,  1.,  2.],
       [ 1.,  1.,  1.,  2.],
       [ 1.,  1.,  1.,  2.]])
```

6.1.6 Indexing and slicing an array

An array is indexed by a tuple of integers, and as for Python sequences negative indexes count from the end of the axis. Slicing and striding is supported in the same way as well. For one-dimensional arrays there is only one index:

```
In [x]: a = np.linspace(1,6,6)
In [x]: print(a)
[ 1.  2.  3.  4.  5.  6.]
In [x]: a[1:4:2]    # elements a[1] and a[3] (a stride of 2)
Out[x]: array([ 2.,  4.])
In [x]: a[3::-2]    # elements a[3] and a[1] (a stride of -2)
Out[x]: array([ 4.,  2.]
```

Multidimensional arrays have an index for each axis. If you want to select every item along a particular axis, replace its index with a single colon:

```
In [x]: a = np.linspace(1,12,12).reshape(4,3)
In [x]: a
Out[x]:
array([[  1.,   2.,   3.],
       [  4.,   5.,   6.],
       [  7.,   8.,   9.],
       [ 10.,  11.,  12.]])
In [x]: a[3, 1]
Out[x]: 11.0
In [x]: a[2, :]         # everything in the third row
Out[x]:
array([ 7.,  8.,  9.])
In [x]: a[:, 1]         # everything in the second column
Out[x]: array([  2.,   5.,   8.,  11.])
In [x]: a[1:-1, 1:]     # middle rows, second column onwards
Out[x]:
```

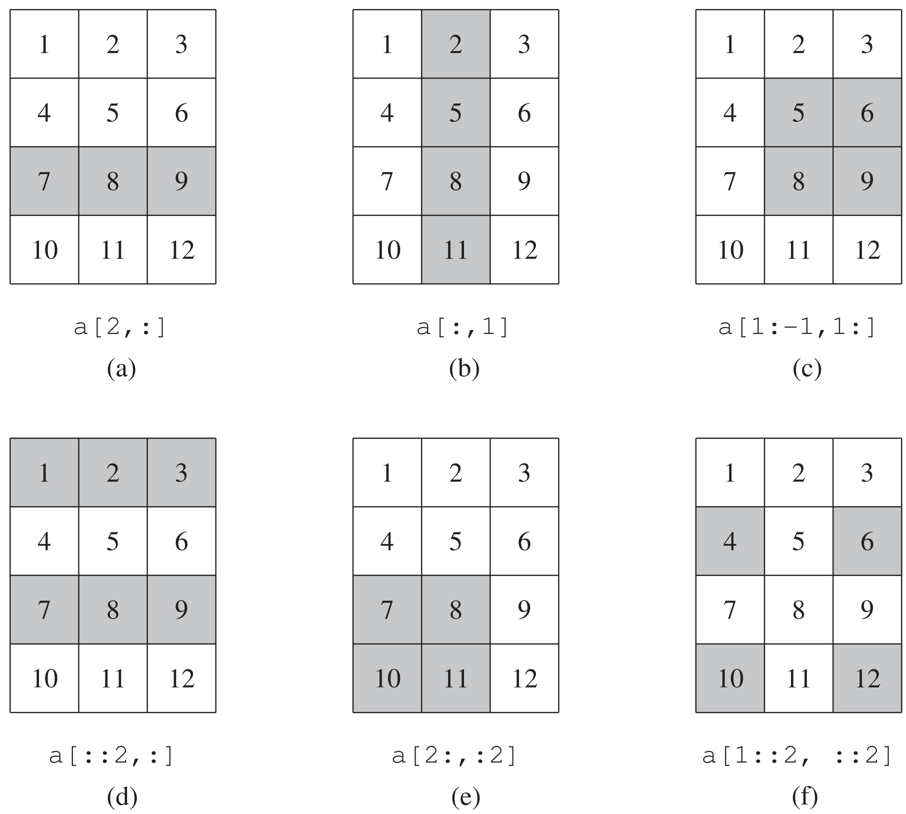


Figure 6.1 Various ways to slice a NumPy array.

```
array([[ 5.,  6.],
       [ 8.,  9.]])
```

These and further examples of NumPy array slicing are illustrated in Figure 6.1.

The special *ellipsis* notation (`...`) is useful for high-rank arrays: in an index, it represents as many colons as are necessary to represent the remaining axes. For example, for a four-dimensional array, `a[3, 1, ...]` is equivalent to `a[3, 1, :, :]` and `a[3,...,1]` is equivalent to `a[3, :, :, 1]`.

The colon and ellipsis syntax also works for assignment:

```
In [x]: a[:,1] = 0      # set all elements in the second column to zero
In [x]: print(a)
[[  1.   0.   3.]
 [  4.   0.   6.]
 [  7.   0.   9.]
 [ 10.   0.  12.]]
```

Advanced indexing

NumPy arrays can also be indexed by sequences that aren't simple tuples of integers, including other lists, arrays of integers and tuples of tuples. Such "advanced indexing" creates a new array with its own copy of the data, rather than a view:

```
In [x]: a = np.linspace(0.,0.5,6)
In [x]: print(a)
```

```
[ 0.   0.1  0.2  0.3  0.4  0.5]
In [x]: ia = [1, 4, 5]      # a list of indexes
In [x]: print(a[ia])
[ 0.1  0.4  0.5]
In [x]: ia = np.array( ((1,2), (3,4)) )
In [x]: print(a[ia])     # an array to be formed from the specified indexes
[[ 0.1  0.2]
 [ 0.3  0.4]]
```

One can even index a multidimensional array with multidimensional arrays of indexes, picking off individual elements at will to build an array of a specified shape. This can lead to some rather baroque code:

```
In [x]: a = np.linspace(1,12,12).reshape(4,3)
In [x]: print(a)
[[  1.   2.   3.]
 [  4.   5.   6.]
 [  7.   8.   9.]
 [ 10.  11.  12.]]
In [x]: ia = np.array( ((1,0),(2,1)) )
In [x]: ja = np.array( ((0,1),(1,2)) )
In [x]: print(a[ia,ja])
[[ 4.  2.]
 [ 8.  6.]]
```

Here we build a 2×2 array (the shape of the index arrays) whose elements are `a[1,0]`, `a[0,1]` on the top row and `a[2,1]`, `a[1,2]` on the bottom row.

Instead of indexing an array with a sequence of integers, it is also possible to use an array of boolean values. The `True` elements of this indexing array identify elements in the target array to be returned:

```
In [x]: a = np.array([-2,-1,0,1,2])
In [x]: ia = np.array([False, True, False, True, True])
In [x]: print(a[ia])
[-1  1  2]
```

Because comparisons are vectorized across arrays just like mathematical operations, this leads to some useful shortcuts:

```
In [x]: print(a)
[-2 -1  0  1  2]
In [x]: ib = a < 0
In [x]: print(ib)
[ True  True False False False]
In [x]: a[ib] = 0   # set all negative elements to zero
In [x]: print(a)
[0 0 0 1 2]
```

It is not actually necessary to store the intermediate boolean array, `ib`, and `a[a<0]=0` does the same job:

```
In [x]: a = np.array([-2,-1,0,1,2])
In [x]: a[a<0]=0
In [x]: print(a)
[0 0 0 1 2]
```

The boolean operations *not*, *and* and *or* are implemented on boolean arrays with the operators ~, & and | respectively. For example,

```
In [x]: years = array([1900, 1904, 1990, 1993, 2000, 2014, 2016, 2100])
In [x]: leap_year = (years % 400 == 0) | (years % 4 == 0) & ~(years % 100 == 0)
In [x]: print(list(zip(years, leap_year)))
Out[x]: [(1900, False), (1904, True), (1990, False), (1993, False),
         (2000, True), (2014, False), (2016, True), (2100, False)]
```

Adding an axis

To add an axis (i.e., dimension) to an array, insert `np.newaxis` in the desired position:

```
In [x]: a = np.linspace(1, 4, 4).reshape(2, 2)
In [x]: print(a)     # a 2x2 array (rank=2)
[[ 1.  2.]
 [ 3.  4.]]
In [x]: a.shape()
(2, 2)
In [x]: b = a[:, np.newaxis, :]
In [x]: print(b)     # a 2x1x2 array (rank=3)
[[[ 1.  2.]]

 [[ 3.  4.]]]
In [x]: b.shape
(2, 1, 2)
```

In fact, `np.newaxis` is the `None` object, so `None` can be used directly in its place if desired.

Example E6.4 A *Sudoku* square consists of a 9×9 grid with entries such that each row, column and each of the 9 nonoverlapping 3×3 tiles contains the numbers 1–9 once only. The following program verifies that a provided grid is a valid Sudoku square.

Listing 6.2 Verifying the validity of a Sudoku square

```
import numpy as np

def check_sudoku(grid):
    """ Return True if grid is a valid Sudoku square, otherwise False. """
    for i in range(9):
        # j, k index the top left-hand corner of each 3x3 tile
        j, k = (i // 3) * 3, (i % 3) * 3
❶       if len(set(grid[i,:])) != 9 or len(set(grid[:,i])) != 9\
                   or len(set(grid[j:j+3, k:k+3].ravel())) != 9:
            return False
    return True

sudoku = """145327698
            839654127
            672918543
            496185372
            218473956
```

```
            753296481
            367542819
            984761235
            521839764"""
# Turn the provided string, sudoku, into an integer array
grid = np.array([[int(i) for i in line] for line in sudoku.split()])
print(grid)

if check_sudoku(grid):
    print('grid valid')
else:
    print('grid invalid')
```

❶ Here we use the fact that an array of length 9 contains nine unique elements if the *set* formed from these elements has cardinality 9. No check is made that the elements themselves are actually the numbers 1–9.

Meshes

To evaluate a multidimensional function on a grid of points, a *mesh* is useful. The function `np.meshgrid` is passed a series of N one-dimensional arrays representing coordinates along each dimension and returns a set of N-dimensional arrays comprising a mesh of coordinates at which the function can be evaluated. For example, in the two-dimensional case:

```
In [x]: x = np.linspace(0, 5, 6)
In [x]: y = np.linspace(0, 3, 4)
In [x]: X, Y = np.meshgrid(x, y)
In [x]: X
Out[x]:
array([[ 0.,  1.,  2.,  3.,  4.,  5.],
       [ 0.,  1.,  2.,  3.,  4.,  5.],
       [ 0.,  1.,  2.,  3.,  4.,  5.],
       [ 0.,  1.,  2.,  3.,  4.,  5.]])

In [x]: Y
Out[x]:
array([[ 0.,  0.,  0.,  0.,  0.,  0.],
       [ 1.,  1.,  1.,  1.,  1.,  1.],
       [ 2.,  2.,  2.,  2.,  2.,  2.],
       [ 3.,  3.,  3.,  3.,  3.,  3.]])
```

The arrays `X` and `Y` can *each* be indexed with indexes `i,j`: the `x` array is repeated as rows down `X` and the `y` array as columns across `Y`. A function of two coordinates can therefore be evaluated on the grid as simply `f(X,Y)`.

Setting the optional argument `sparse` to `True` will return sparse grid to conserve memory. In the previous example, instead of two arrays, both with shapes `(6, 4)`, arrays with shapes `(1,6)` and `(4, 1)` that can be broadcast against each other (see Section 6.1.7) will be returned:

```
In [X]: X, Y = np.meshgrid(x, y, sparse=True)
In [X]: X
Out[X]: array([[ 0.,  1.,  2.,  3.,  4.,  5.]])
```

```
In [X]: Y
Out[X]:
array([[ 0.],
       [ 1.],
       [ 2.],
       [ 3.]])
```

6.1.7 ◇ Broadcasting

We have already seen that simple operations such as addition and multiplication can be carried out elementwise on two arrays of the same shape (*vectorization*):

```
In [x]: a = np.array([1, 2, 3])
In [x]: b = np.array([0, 10, 100])
In [x]: a * b
Out[x]: array([  0,  20, 300])
```

Broadcasting describes the rules that NumPy uses to carry out such operations when the arrays have *different* shapes. This allows the operation to be carried out using precompiled C loops instead of slower, Python loops, but there are constraints as to which array shapes can be broadcast against each other. The rules are applied on each dimension of the arrays, starting with the last and working backward. Two dimensions compared in this way are said to be *compatible* if they are *equal* or *one of them is 1*.

The simplest example of broadcasting involves the operation between an array and a scalar (which may be considered for this purpose to be a one-dimensional array of length 1). Consider

```
In [x]: a = np.array([[1, 2, 3], [4, 5, 6]])
In [x]: b = 2
In [x]: c = a * b
In [x]: c
Out[x]:
array([[ 2,  4,  6],
       [ 8, 10, 12]])
```

The dimensions of `a` and `b` are compatible:

```
a:      2 x 3
b:          1
c:      2 x 3
```

Here, `b` can be broadcast across the two dimensions of array `a` by repetition of its value for every element in that array. Similarly, an array of shape `(3,)` can be broadcast across both rows of `a`:

```
In [x]: b = np.array([1, 2, 3])
In [x]: a*b
Out[x]:
array([[ 1,  4,  9],
       [ 4, 10, 18]])

a:      2 x 3
b:          3
c:      2 x 3
```

That is, for each row of a, its entries are multiplied by the corresponding entries of the one-dimensional array b. However, attempting to multiply a by an array whose last dimension is not 1 or 3 is a ValueError here:

```
In [x]: b = np.array([1,2])
In [x]: a * b


---------------------------------------------------------------------------
...
----> 1 a * b

ValueError: operands could not be broadcast together with shapes (2,3) (2,)
```

In the example of the sparse mesh created in the previous section, the arrays with shapes (1,6) and (4,1) are compatible. For example,

```
In [x]: f = X*Y
Out[x]: f
array([[  0.,    0.,    0.,    0.,    0.,    0.],
       [  0.,    1.,    2.,    3.,    4.,    5.],
       [  0.,    2.,    4.,    6.,    8.,   10.],
       [  0.,    3.,    6.,    9.,   12.,   15.]])
```

The broadcasting process “stretches out” the second axis of Y from 1 to 6 to match that of X and the first axis of X from 1 to 4 to match that of Y:

```
X:      1 x 6
Y:      4 x 1
f:      4 x 6
```

To force a broadcast on an array with insufficient dimensions to meet your requirements, you can always add an axis with np.newaxis. For example, one way to take the *outer product* of two arrays is by adding a dimension to one of them and broadcasting the multiplication:

```
In [x]: a = np.array([1, 2, 3])
In [x]: b = np.array([0, 10, 100])
In [x]: c = a[:, np.newaxis] * b
In [x]: c
Out[x]:
array([[  0,  20, 300],
       [  0,  40, 600],
       [  0,  60, 900]])
```

Thus, instead of matching elements in the two arrays with shapes (3,), the extra axis on a creates an array with shape (3,1) and this dimension is stretched across the array b:

```
a[:,np.newaxis]:   3 x 1
              b:       3
              c:   3 x 3
```

6.1.8 Maximum and minimum values

NumPy arrays have the methods `min` and `max`, which return the minimum and maximum values in the array. By default, a single value for the flattened array is returned; to find maximum and minimum values along a given axis, use the `axis` argument:

```
In [x]: a = np.array([[3, 0, -1, 1], [2, -1, -2, 4], [1, 7, 0, 4]])
In [x]: print(a)
[[ 3  0 -1  1]
 [ 2 -1 -2  4]
 [ 1  7  0  4]]
In [x]: a.min()     # "global" minimum
Out[x]: -2
In [x]: a.max()     # "global" maximum
Out[x]: 7
In [x]: print( a.min(axis=0) )
[ 1 -1 -2  1]       # minima in each column
In [x]: print( a.max(axis=1) )
[3 4 7]             # maxima in each row
```

Often one wants not the maximum (or minimum) value itself but its index in the array. This is what the methods `argmin` and `argmax` do. By default, the index returned is into the *flattened* array, so the actual value can be retrieved using a view on the array created by `ravel`:

```
In [x]: a.argmin()
6
In [x]: a.ravel()[a.argmin()]
-2
In [x]: print(a.argmax(axis=0))
[0 2 2 1]       # row indexes of maxima in each column
In [x]: print(a.argmax(axis=1))
[0 3 1]         # column indexes of maxima in each row
```

Figure 6.2 illustrates the process for `axis=0` and for `axis=1`. Notice that if more than one equal maximum exists in a column, the index of the first is returned.

Example E6.5 Consider the following oscillating functions on the interval $[0, L]$:

$$f_n(x) = x(L-x)\sin\frac{2\pi x}{\lambda_n}; \quad \lambda_n = \frac{2L}{n}, \quad n = 1, 2, 3, \cdots .$$

The following code defines a two-dimensional array holding values of these functions for $L = 1$ on a grid of $N = 100$ points (rows) for $n = 1, 2, \cdots, 5$ (columns). The position of the maximum and minimum in each column is calculated with `argmax(axis=0)` and `argmin(axis=0)`. (See Figure 6.3.)

Listing 6.3 `argmax` and `argmin`

```
# eg6-array_maxmin.py
import numpy as np
import pylab

N = 100
L = 1
```

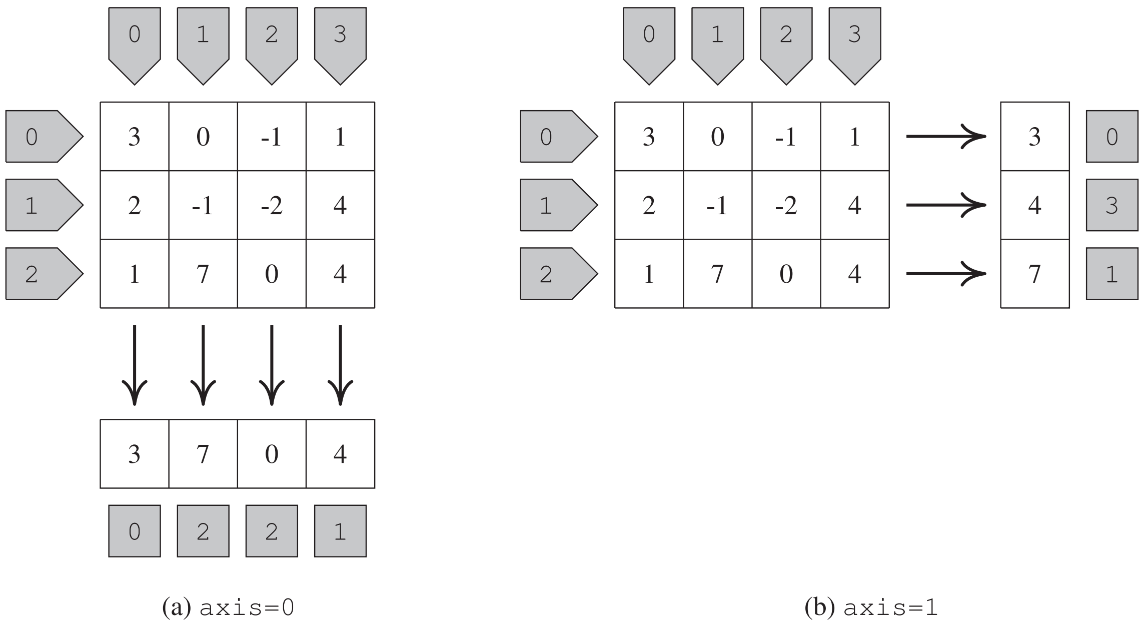


Figure 6.2 (a) `a.max(axis=0)` giving the maximum values and `a.argmax(axis=0)` giving the indexes of the maximum values of each column in array `a` (that is, maintaining the *row* dimension) and (b) The same for `axis=1`: maximum values along each row.

```
def f(i, n):
    x = i * L / N
    lam = 2*L/(n+1)
    return x * (L-x) * np.sin(2*np.pi*x/lam)

a = np.fromfunction(f, (N+1, 5))
min_i = a.argmin(axis=0)
max_i = a.argmax(axis=0)
pylab.plot(a, c='k')
pylab.plot(min_i, a[min_i, np.arange(5)], 'v', c='k', markersize=10)
pylab.plot(max_i, a[max_i, np.arange(5)], '^', c='k', markersize=10)
pylab.xlabel(r'$x$')
pylab.ylabel(r'$f_n(x)$')
pylab.show()
```

6.1.9 Sorting an array

NumPy arrays can be sorted in several different ways with the `sort` method, which orders the numbers in an array *in place*. By default, this method sorts multidimensional arrays along their *last* axis. To sort along some other axis, set the `axis` argument. For example,

```
In [x]: a = np.array([5, -1, 2, 4, 0, 4])
In [x]: a.sort()
```

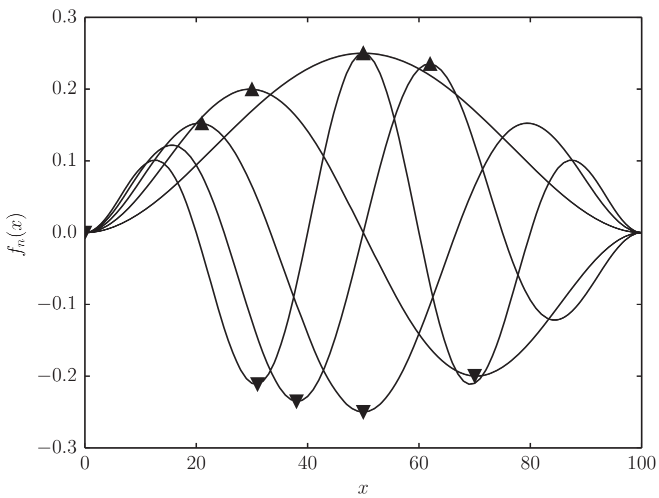


Figure 6.3 Maxima and minima of the functions $f_n(x)$ described in Example E6.5. Note that only the "global" maximum and minimum are returned for each function, and that where more than one point has the same maximum or minimum value, only the first is returned.

```
In [x]: print(a)
[-1  0  2  4  4  5]
In [x]: b = np.array([[0, 3, -2], [7, 1, 3], [4, 0, -1]])
In [x]: print(b)
[[ 0  3 -2]
 [ 7  1  3]
 [ 4  0 -1]]
In [x]: b.sort()          # sort the numbers along each row
In [x]: print(b)
[[-2  0  3]
 [ 1  3  7]
 [-1  0  4]]
```

This is the same as `b.sort(axis=1)` – "for each row, order the numbers by column." To sort the numbers in each column – "for each column, order the numbers by row," set `axis=0`:

```
In [x]: b.np.array([[0, 3, -2], [7, 1, 3], [4, 0, -1]])
In [x]: b.sort(axis=0)    # sort the numbers along each column
In [x]: print(b)
[[0  0  -2]
 [4  1  -1]
 [7  3  3]]
```

The sorting algorithm used is the "quicksort" algorithm, which is a good general-purpose choice.[11]

[11] Some arrays can be sorted faster with the alternative *mergesort* or *heapsort* algorithms; these can be selected by setting the optional `kind` argument to the string literal values `'mergesort'` and `'heapsort'`, for example: `b.sort(axis=1, kind='heapsort')`.

Two other sorting functions are worth mentioning. `np.argsort` returns the *indexes* that would sort an array rather than the sorted elements themselves:

```
In [x]: a = np.array([3, 0, -1, 1])
In [x]: np.argsort(a)
Out[x]: array([2, 1, 3, 0])
```

Therefore,

```
In [x]: a[np.argsort(a)]
Out[x]: array([-1,  0,  1,  3])
```

`np.argsort` also takes the `axis` and `kind` arguments previously described.

The method `np.searchsorted` takes a, *sorted* array, `a` and one or more values, `v`, and returns the indexes in `a` at which the values should be entered to maintain its order:

```
In [x]: a = np.array([1, 2, 3, 4])
In [x]: np.searchsorted(a, 3.5)
Out[x]: 3
In [x]: np.searchsorted(a, (3.5, 0, 1.1))
Out[x]: array([3, 0, 1])
```

6.1.10 Structured arrays

Also known as *record arrays*, structured arrays are arrays consisting of rows of values where each value may have its own data type and name. These rows are the "records." This is very much like a table of data with rows (records) consisting of values that fall into columns (fields) and provides a very convenient and natural way to manipulate scientific data that is often obtained or presented in tabular form.

Creating a structured array

The structure of a record array is defined by its `dtype` using a more complex syntax than we have used previously. For example,

```
In [x]: a = np.zeros(5, dtype='int8, float32, complex_')
In [x]: print(a)
[(0, 0.0, 0j) (0, 0.0, 0j) (0, 0.0, 0j) (0, 0.0, 0j) (0, 0.0, 0j)]
In [x]: a.dtype
dtype([('f0', '|i1'), ('f1', '<f4'), ('f2', '<c16')])
```

Here we have created an array of five records, each of which has three fields, defined by constructing a `dtype` specified by the string `'int8, float32, complex_'`.

- The first field is a single-byte, signed integer (`int8` which is described by the string `'|i1'` – clearly the endianness (byte order) is not relevant in a one-byte quantity);
- The second is a single-precision floating point number (which on my system) is stored in memory as a little-endian 4-byte sequence, indicated by `'<f4'`;
- The final field is defined to be a complex number to default precision, which on my system is stored in 16-bytes, little-endian (`complex_` is equivalent to `complex128` which corresponds to a data type `'<c16'`).

Because we did not explicitly name the fields, they are given the default names `'f0'`, `'f1'` and `'f2'`. To name the fields of our structured array explicitly, pass the `dtype` constructor a list of `(name, dtype descriptor)` tuples: for example,

```
In [x]: dt = np.dtype( [('time', 'f8'), ('signal', 'i4')] )
In [x]: a = np.zeros(10, dtype=dt)
In [x]: a
Out[x]:
array([(0.0, 0), (0.0, 0), ..., (0.0, 0)],
      dtype=[('time', '<f8'), ('signal', '<i4')])
```

A structured array can therefore be visualized as a table of data values with column headings for each field.

Assigning records in a structured array is as expected:

```
In [x]: a[0] = (0., 4)
In [x]: a[1:3] = [(0.5, -3), (1., -5)]
In [x]: a
Out[x]:
array([(0.0, 4), (0.5, -3), (1.0, -5), ..., (0.0, 0)],
      dtype=[('time', '<f8'), ('signal', '<i4')])
```

but the real power of this approach is in the ability to reference a field by its name. For example, to set the `'time'` column in our array to a linear sequence:

```
In [x]: a['time'] = np.linspace(0., 4.5, 10)
In [x]: print(a)
[(0.0, 4) (0.5, -3) (1.0, -5) (1.5, 0) (2.0, 0) (2.5, 0) (3.0, 0) (3.5, 0)
 (4.0, 0) (4.5, 0)]
In [x]: print(a['time'][-1])
4.5
```

Likewise, to obtain a view on a column, refer to it by name:

```
In [x]: print(a['time'])
[ 0.   0.5  1.   1.5  2.   2.5  3.   3.5  4.   4.5]
In [x]: print( a['signal'].min() )
-5
```

More ways to create a structured array

There are several (arguably, too many) ways to define the `dtype` describing a structured array. So far we have used a string of comma-separated identifiers and a list of tuples. A third way is to use a *dictionary*. The basic usage assigns a list of values to the two keys `'names'` and `'formats'` naming the fields and specifying their formats respectively:

```
In [x]: dt = np.dtype({ 'names': ['time', 'signal'],
                        'formats':['f8', 'i4']
                      })
In [x]: a = np.zeros(10, dtype=dt)
```

defines the same structured array of `(time, signal)` records as before. A third key, `'titles'`, can be used to give each field a more detailed description; each title can then be used as an alias to its name in referring to that field in the array.[12]

[12] In fact, `title` can be any Python object and can be used to provide detailed "metadata" concerning the corresponding field.

```
In [x]: dt = np.dtype({'names': ['candidate', 'mark', 'grade'],
                'formats': ['|S50', 'u1', '|S2'],
                'titles': ['Candidate Name', 'Percentage Mark', 'Grade: A-F']})
In [x]: a = np.zeros(10, dtype=dt)
In [x]: a[0] = ('John Brown', 64, 'B-')
In [x]: a[1] = ('Jane Smith', 78, 'A')
In [x]: print(a['Candidate Name'])
[b'John Brown' b'Jane Smith' b'' b'' b'' b'' b'' b'' b'' b'']
In [x]: print(a['Percentage Mark'])
[64 78  0  0  0  0  0  0  0  0]
```

Sorting structured arrays

Structured arrays can be sorted by giving a specific order to the fields used with the `order` argument. For example, with the following structured array:

```
In [x]: data = [ ('NiCd', 1.2, 0.14, 2000),
                 ('Lead acid', 2.1, 0.14, 700),
                 ('Lithium ion', 3.6, 0.46, 800) ]
In [x]: dtype = [ ('name', '|S20'),
                  ('voltage', 'f8'),
                  ('specific energy', 'f8'),
                  ('cycle durability', 'i4') ]
In [x]: a = np.array(data, dtype=dtype)
In [x]: a.sort(order='specific energy')
In [x]: print(a)
[(b'Lead acid', 2.1, 0.14, 700) (b'NiCd', 1.2, 0.14, 2000)
 (b'Lithium ion', 3.6, 0.46, 800)]

In [x]: a.sort(order=['specific energy', 'voltage'])
In [x]: print(a)
[(b'NiCd', 1.2, 0.14, 2000) (b'Lead acid', 2.1, 0.14, 700)
 (b'Lithium ion', 3.6, 0.46, 800)]
```

The second sort operation here sorts the records by specific energy, and if this is the same for two or more records, then it sorts by voltage.

6.1.11 Arrays as vectors

Although NumPy provides a `matrix` class that specializes `ndarray` to make linear algebra calculations easier and can be used to represent vectors, for many purposes it is just as convenient to define a vector with n components as a regular one-dimensional array with `n` elements.

In addition to elementwise operations such as vector addition, subtraction and so on, NumPy array objects implement scalar (dot) product and vector (cross) product methods:

```
In [x]: a = np.array([1, 0, -3])      # vector as a one-dimensional array
In [x]: b = np.array([2, -2, 5])
In [x]: a.dot(b)                      # or b.dot(a) or np.dot(a,b)
Out[x]: -13
In [x]: np.cross(a, b)
array([ -6, -11,  -2])
```

You can only take the cross product of an array with two or three elements; the third component is assumed to be zero in the former case. To use `dot` and `cross` on two individual vectors, ensure that they are row vectors as described previously and not column vectors represented as an `(n,1)` array:

```
In [x]: a = np.array([[1], [0],[-3]])   # 3x1 two-dimensional array
In [x]: b = np.array([[2],[-2], [5]])
In [x]: print(a)
[[ 1]
 [ 0]
 [-3]]
In [x]: np.dot(a,b)     # tries matrix multiplication: won't work
...

ValueError: objects are not aligned
```

If you do want to take the dot product of two column vectors using `np.dot`, they need to be turned into row vectors:

```
In [x]: np.dot(a.T[0], b.T[0])    # transpose to row vectors
Out[x]: -13
```

This is a bit tortuous: the index is needed because the transpose of our `(n,1)` (two-dimensional) array is a `(1,n)` array from which we want the first and only row for our vector. Alternatively, we can operate using a flattened view of the column vectors obtained with `ravel`:

```
In [x]: a.ravel().dot(b.ravel())
Out[x]: -13
```

See also Section 6.6.

6.1.12 Logic and comparisons

NumPy provides a set of methods for comparing and performing logical operations on arrays elementwise. The more useful of these are summarized in Table 6.4.

Table 6.4 `ndarray` Attributes

Function	Description
`np.all(a)`	Determine whether *all* array elements of `a` evaluate to `True`.
`np.any(a)`	Determine whether *any* array element of `a` evaluates to `True`.
`np.isreal(a)`	Determine whether each element of array `a` is real.
`np.iscomplex(a)`	Determine whether each element of array `a` is a complex number.
`np.isclose(a, b)`	Return a boolean array of the comparison between arrays `a` and `b` for equality within some tolerance.
`np.allclose(a, b)`	Return a `True` if *all* the elements in the arrays `a` and `b` are equal to within some tolerance.

`np.all` and `np.any` work the same as Python's built-in functions of the same name[13] (see Section 2.4.3):

```
In [x]: a = np.array([[1, 2, 0, 3], [4, 0, 1, 1]])
In [x]: np.any(a), np.all(a)
Out[x]: (True, False)     # Some (but not all) elements are equivalent to True
```

`np.isreal` and `np.iscomplex` return boolean arrays:

```
In [x]: b = np.array([1, -1j, 0.5j, 0, 1-2.5j])
In [x]: np.isreal(b)
Out[x]: array([ True, False, False,  True, False], dtype=bool)
In [x]: np.iscomplex(b)
Out[x]: array([False,  True,  True, False,  True], dtype=bool)
```

Because the representation of floating point numbers is not exact, comparing two `float` or `complex` arrays with the `==` operator is not always reliable and is not recommended. Instead, the best we can do is see if two values are "close" to one another within some (typically small) absolute or relative tolerance – NumPy provides the function `np.isclose(a, b)` for elementwise comparisons of two arrays: it returns `True` for elements satisfying

```
abs(a-b) <= (atol + rtol * abs(b))
```

with absolute tolerance, `atol` and relative tolerance, `rtol` which are 10^{-8} and 10^{-5} respecively by default but can be changed by setting the corresponding arguments.[14] An additional argument, `equal_nan`, defaults to `False`, meaning that `nan` values in corresponding positions in the two arrays are treated as different; to treat such elements as equal, set `equal_nan=True`.

```
In [x]: a = np.array([1.66e-27, 1.38e-23, 6.63e-34, 6.02e23, np.nan])
In [x]: b = np.array([1.66e-27, 1.66e-27, 1.66e-27, 6.00e23, np.nan])
In [x]: np.isclose(a, b)
Out[x]: array([ True,  True,  True, False, False], dtype=bool)
In [x]: np.isclose(a, b, equal_nan=True)
Out[x]: array([ True,  True,  True, False,  True], dtype=bool)
```

Note that small numbers compare as equal even though they may differ by many orders of magnitude – to correct this, set `atol=0` to compare within relative tolerance only:

```
In [x]: np.isclose(a, b, atol=0)
Out[x]: array([ True, False, False, False, False], dtype=bool)
```

Finally, `allclose(a, b)` returns a single value: `True` only if every element in `a` is equal to the corresponding element in `b` (within the tolerance defined by `atol` and `rtol`), and otherwise `False`.

```
In [x]: x = np.linspace(0, np.pi, 100)
In [x]: np.allclose(np.sin(x)**2, 1 - np.cos(x)**2)
Out[x]: True
```

[13] Except that they don't work on generator or iterator objects.

[14] Note that this relation is not symmetric in `a` and `b`, so it is possible that `isclose(a, b)` may not equal `isclose(b, a)`.

6.1.13 Exercises

Questions

Q6.1.1 What is the difference between the objects `np.ndarray` and `np.array`?

Q6.1.2 Why doesn't this create a two-dimensional array?

```
>>> np.array((1,0,0), (0,1,0), (0,0,1), dtype=float)
```

What is the correct way?

Q6.1.3 What is the difference, if any, between the following statements:

```
>>> a = np.array([0,0,0])
>>> a = np.array([[0,0,0]])
```

Q6.1.4 Explain the following behavior:

```
In [x]: a, b = np.zeros((3,)), np.ones((3,))
In [x]: a.dtype = 'int'
In [x]: a
Out[x]: array([0, 0, 0])
In [x]: b.dtype = 'int'
In [x]: b
Out[x]: array([4607182418800017408, 4607182418800017408, 4607182418800017408])
```

What is the correct way to convert an array of one data type to an array of another?

Q6.1.5 A $3 \times 4 \times 4$ array is created with

```
In [x]: a = np.linspace(1,48,48).reshape(3,4,4)
```

Index or slice this array to obtain the following:

a. `20.0`

b. `[ 9. 10. 11. 12.]`

c. The 4×4 array:

```
[[ 33.  34.  35.  36.]
 [ 37.  38.  39.  40.]
 [ 41.  42.  43.  44.]
 [ 45.  46.  47.  48.]]
```

d. The 3×2 array:

```
[[  5.,   6.],
 [ 21.,  22.],
 [ 37.,  38.]]
```

e. The 4×2 array:

```
[[ 36.  35.]
 [ 40.  39.]
 [ 44.  43.]
 [ 48.  47.]]
```

f. The 3×4 array:

```
[[ 13.   9.   5.   1.]
 [ 29.  25.  21.  17.]
 [ 45.  41.  37.  33.]]
```

g. (Harder) Using an array of indexes, the 2×2 array:

```
[[  1.   4.]
 [ 45.  48.]]
```

Q6.1.6 Write an expression, using boolean indexing, which returns only the values from an array that have magnitudes between 0 and 1.

Q6.1.7 Why does the following statement evaluate to `True` even though the two numbers passed to `np.isclose()` differ by more than `atol`?

```
In [x]: np.isclose(-2.00231930436153, -2.0023193043615, atol=1.e-14)
Out[x]: True
```

Q6.1.8 Explain why the following evaluates to `True` even though the two approximations to π differ by more than 10^{-16}:

```
In [x]: np.isclose(3.1415926535897932, 3.141592653589793, atol=1.e-16, rtol=0)
Out[x]: True
```

whereas this statement works as expected:

```
In [x]: np.isclose(3.14159265358979, 3.1415926535897, atol=1.e-14, rtol=0)
Out[x]: False
```

Q6.1.9 Verify that the magic square created in Example E6.2 satisfies the conditions that it contains the numbers 1 to N^2 and that its rows, columns and main diagonals sum to $N(N^2 + 1)/2$.

Q6.1.10 Write a one-line statement that returns `True` if an array is a monotonically increasing sequence or `False` otherwise.

Hint: `np.diff` returns the *difference* between consecutive elements of a sequence. For example,

```
In [x]: np.diff([1,2,3,3,2])
Out[x]: array([ 1,  1,  0, -1])
```

◊ **Q6.1.11** (Harder) The `dtype` `np.uint8` represents an unsigned integer in 8 bits. Its value may therefore be in the range 0–255. Explain the following behavior:

```
In [x]: x = np.uint8(250)
In [x]: x*2
Out[x]: 500

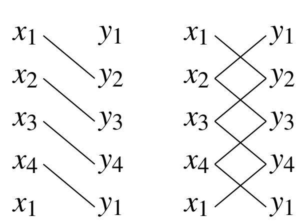
In [x]: x = np.array([250,], dtype=np.uint8)
In [x]: x*2
Out[x]: array([244], dtype=uint8)
```

Problems

P6.1.1 Turn the following data concerning various species of cetacean into a NumPy structured array and order it by (a) mass and (b) population. Determine in each case the index at which *Bryde's whale* (population: 100000, mass: 25 tonnes) should be inserted to keep the array ordered.

Name	Population	Mass/tonnes
Bowhead whale	9000	60
Blue whale	20000	120
Fin whale	100000	70
Humpback whale	80000	30
Gray whale	26000	35
Atlantic white-sided dolphin	250000	0.235
Pacific white-sided dolphin	1000000	0.15
Killer whale	100000	4.5
Narwhal	25000	1.5
Beluga	100000	1.5
Sperm whale	2000000	50
Baiji	13	0.13
North Atlantic right whale	300	75
North Pacific right whale	200	80
Southern right whale	7000	70

P6.1.2 The *shoelace algorithm* for calculating the area of a simple polygon (that is, one without holes or self-intersections) proceeds as follows: Write down the (x, y) coordinates of the N vertices in an $N \times 2$ array and then repeat the coordinates of the first vertex as the last row to make an $(N + 1) \times 2$ array. Now (a) multiply each x-coordinate value in the first N rows by the y-coordinate value in the next row down and take the sum, $S_1 = x_1y_2 + x_2y_3 + \cdots + x_Ny_1$. Then (b) multiply each y-coordinate value in the first N rows by the x-coordinate in the next row down and take the sum, $S_2 = y_1x_2 + y_2x_3 + \cdots + y_Nx_1$. The area of the polygon is then $\frac{1}{2}|S_1 - S_2|$.



(a) (b)

Implement this algorithm as a function that takes a NumPy array of vertices as its argument and returns the area of the polygon. Do not use Python loops!

P6.1.3 Using NumPy, it is possible to do this exercise without using a single (Python) loop.

The normalized Gaussian function with mean μ and standard deviation σ is

$$g(x) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left(\frac{(x-\mu)^2}{2\sigma^2}\right).$$

Write a program to calculate and plot the Gaussian functions with $\mu = 0$ and the three values $\sigma = 0.5, 1, 1.5$. Use a grid of 1,000 points in the interval $-10 \le x \le 10$.

Verify (by direct summation) that the functions are normalized with area 1.

Finally, calculate the first derivative of these functions on the same grid using the first-order central difference approximation:

$$g'(x) \approx \frac{g(x+h) - g(x-h)}{2h}$$

for some suitably chosen, small h.

6.2 Reading and writing an array to a file

Scientific data are frequently read in from a text file, which may contain comments, missing values and blank lines. Columns of values may be either aligned in a fixed-width format or separated by one or more delimiting characters (such as spaces, tabs or commas). Furthermore, there may be a descriptive header and even footnotes to the file, which make it hard to parse directly using Python's string methods.

NumPy provides several functions for reading data from a text file. The simpler `np.loadtxt` handles many common cases; the more sophisticated `np.genfromtxt` allows for better handling of missing values and footers. These are described in the following sections.

6.2.1 `np.save` and `np.load`

There is a platform-independent *binary* format for saving a NumPy array:

```
In [x]: np.save('my-array.npy', a)
```

will save the array `a` to the binary file `my-array.npy` (the `.npy` extension is appended if it is not provided). The array can then be reloaded using NumPy on any other operating system with

```
In [x]: a = np.load('my-array.npy')
```

(the `.npy` extension must be provided).

6.2.2 `np.loadtxt`

The method prototype for `np.loadtxt` is

```
np.loadtxt(fname, dtype=<class 'float'>, comments='#',
           delimiter=None, converters=None, skiprows=0,
           usecols=None, unpack=False, ndmin=0)
```

The arguments are as follows:

- `fname`: The only required argument, `fname`, which can be a filename, an open file, or a generator returning the lines of data to be parsed.
- `dtype`: The data type of the array defaults to `float` but can be set explicitly by the `dtype` argument. In particular this is the place to set up names and types for a structured array (see Section 6.1.10).
- `comments`: Comments in a file are usually started by some character such as `#` (as with Python) or `%`. To tell NumPy to ignore the contents of any line following this character, use the `comments` argument – by default it is set to `#`.
- `delimiter`: The string used to separate columns of data in the file; by default it is `None`, meaning that any amount of whitespace (spaces, tabs) delimits the data. To read a comma-separated (csv) file, set `delimiter=','`.
- `converters`: An optional dictionary mapping the column index to a function converting string values in that column to data (e.g., `float`).
- `skiprows`: An integer giving the number of lines at the start of the file to skip over before reading the data (e.g., to pass over header lines). Its default is 0 (no header).
- `usecols`: A sequence of column indexes determining which columns of the file to return as data; by default it is `None`, meaning all columns will be parsed and returned.
- `unpack`: By default, the data table is returned in a single array of rows and columns reflecting the structure of the file read in. Set `unpack=True` will transpose this array so that individual columns can be picked off an assigned to different variables.
- `ndmin`: The minimum number of dimensions the returned array should have. By default, 0 (so a file containing a single number is read in as a scalar), it can be set to 1 or 2.

For example, to read the first, third and fourth columns from the file `data.txt` into three separate one-dimensional arrays:

```
col1, col3, col4 = np.loadtxt('data.txt', usecols=(0,2,3), unpack=True)
```

Example E6.6 The use of `np.loadtxt` is best illustrated using an example. Consider the following text file of data relating to a (fictional) population of students. This file can be downloaded as `eg6-a-student-data.txt` from scipython.com/eg/aac.

```
# Student data collected on 17 July 2014
# Researcher: Dr Wicks, University College Newbury

# The following data relate to N = 20 students. It
# has been totally made up and so therefore is 100%
# anonymous.

Subject Sex    DOB      Height  Weight       BP     VO2max
(ID)     M/F  dd/mm/yy     m       kg        mmHg  mL.kg-1.min-1
JW-1      M    19/12/95    1.82     92.4     119/76   39.3
```

```
JW-2     M    11/1/96     1.77     80.9    114/73    35.5
JW-3     F    2/10/95     1.68     69.7    124/79    29.1
JW-6     M    6/7/95      1.72     75.5    110/60    45.5
# JW-7   F    28/3/96     1.66     72.4    101/68    -
JW-9     F    11/12/95    1.78     82.1    115/75    32.3
JW-10    F    7/4/96      1.60     -       -         30.1
JW-11    M    22/8/95     1.72     77.2    97/63     48.8
JW-12    M    23/5/96     1.83     88.9    105/70    37.7
JW-14    F    12/1/96     1.56     56.3    108/72    26.0
JW-15    F    1/6/96      1.64     65.0    99/67     35.7
JW-16    M    10/9/95     1.63     73.0    131/84    29.9
JW-17    M    17/2/96     1.67     89.8    101/76    40.2
JW-18    M    31/7/96     1.66     75.1    -         -
JW-19    F    30/10/95    1.59     67.3    103/69    33.5
JW-22    F    9/3/96      1.70     -       119/80    30.9
JW-23    M    15/5/95     1.97     89.2    124/82    -
JW-24    F    1/12/95     1.66     63.8    100/78    -
JW-25    F    25/10/95    1.63     64.4    -         28.0
JW-26    M    17/4/96     1.69     -       121/82    39.
```

Let's find the average heights of the male and female students. The columns we need are the second and fourth, and there's no missing data in these columns so we can use `np.loadtxt`. First construct a record `dtype` for the two fields, then read the relevant columns after skipping the first nine header lines:

```
In [x]: fname = 'eg6-a-student-data.txt'
In [x]: dtype1 = np.dtype([('gender', '|S1'), ('height', 'f8')])
In [x]: a = np.loadtxt(fname, dtype=dtype1, skiprows=9, usecols=(1,3))
In [x]: a
Out[x]:
array([(b'M', 1.8200000524520874), (b'M', 1.7699999809265137),
       (b'F', 1.6799999475479126), (b'M', 1.7200000286102295),
       ...
       (b'M', 1.690000057220459)],
      dtype=[('gender', 'S1'), ('height', '<f8')])
```

To find the average heights of the male students, we only want to index the records with the gender field as `M`, for which we can create a boolean array:

```
In [x]: m = a['gender'] == b'M'
In [x]: m
Out[x]: array([ True,  True, False,  True, ...,  True], dtype=bool)
```

`m` has entries that are `True` or `False` for each of the 19 valid records (one is commented out) according to whether the student is male or female. So the heights of the male students can be seen to be:

```
In [x]: print(a['height'][m])
[ 1.82000005  1.76999998  1.72000003  1.72000003  1.83000004  1.63
  1.66999996  1.65999997  1.97000003  1.69000006]
```

Therefore, the averages we need are

```
In [x]: m_av = a['height'][m].mean()
In [x]: f_av = a['height'][~m].mean()   ❶
In [x]: print('Male average: {:.2f} m, Female average: {:.2f} m'.format(m_av,f_av))
Male average: 1.75 m, Female average: 1.65 m
```

❶ Note that `~m` ("not `m`") is the inverse boolean array of `m`.

To perform the same analysis on the student weights we have a bit more work to do because there are some missing values (denoted by '`-`'). We could use `np.genfromtxt` (see Section 6.2.3), but let's write a converter method instead. We'll replace the missing values with the nicely unphysical value of −99. The function `parse_weight` expects a string argument and returns a `float`:

```
def parse_weight(s):
    try:
        return float(s)
    except ValueError:
        return -99.
```

This is the function we want to pass as a converter for column 4:

```
In [x]: dtype2 = np.dtype([('gender', '|S1'), ('weight', 'f8')])
In [x]: b = np.loadtxt(fname, dtype=dtype2, skiprows=9, usecols=(1,4),
                       converters={4: parse_weight})
```

Now mask off the invalid data and index the array with a boolean array as before:

```
In [x]: mv = b['weight'] > 0    # elements only True for valid data
In [x]: m_wav = b['weight'][mv & m].mean()      # valid and male
In [x]: f_wav = b['weight'][mv & ~m].mean()     # valid and female
In [x]: print('Male average: {:.2f} kg,
               Female average: {:.2f} kg'.format(m_wav,f_wav))
Male average: 82.44 kg, Female average: 66.94 kg
```

Finally, let's read in the blood pressure data. Here we have a problem, because the systolic and diastolic pressures are not separated by whitespace but by a forward slash (/). One solution is to reformat each line to replace the slash with a space before it is fed to `np.loadtxt`. Recall that `fname` can be a generator instead of a filename or open file: we write a suitable generator function, `reformat_lines`, which takes an open file object and yields its lines to `np.loadtxt`, one by one, after the replacement. This is going to mess with the column numbering because it has the side effect of splitting up the birth dates into three columns, so in our reformatted lines the blood pressure values are now in the columns indexed at 7 and 8.

Listing 6.4 Reading the blood pressure column

```
# eg6-a-read-bp.py
import numpy as np

fname = 'eg6-a-student-data.txt'
dtype3 = np.dtype([('gender', '|S1'), ('bps', 'f8'), ('bpd', 'f8')])

def parse_bp(s):
    try:
        return float(s)
    except ValueError:
        return -99.

def reformat_lines(fi):
    for line in fi:
        line = line.replace('/',' ')
        yield line
```

```
with open(fname) as fi:
    gender, bps, bpd = np.loadtxt(reformat_lines(fi), dtype3, skiprows=9,
                usecols=(1,7,8),converters={7: parse_bp, 8: parse_bp},
                unpack=True)

# now do something with the data...
```

6.2.3 np.genfromtxt

NumPy's genfromtxt function is similar to np.loadtxt but has a few more options and is able to cope with missing data.

The following arguments to this function are the same as for np.loadtxt: fname (the only required argument), dtype, comments, converters, usecols and unpack.

Headers and footers

Instead of np.loadtxt's skiprows, the np.genfromtxt function has two optional arguments, skip_header and skip_footer, giving the number of lines to skip at the beginning and the end of the file, respectively.

Fixed-width fields

The delimiter argument works the same as for np.loadtxt but can also be provided as a sequence of integers giving the widths of each field to be read in where the data does not have delimiters. For example, suppose the following text file, data.txt, is to be interpreted as consisting of four columns with widths 2, 1, 9 and 3 characters:

```
 12  100.231.03
 11 1201.842.04
 11   99.324.02
```

so that the first row is to be split: ' 1', '2', ' 100.231', '.03'. There is no delimiter character, so this isn't possible with np.loadtxt, but with np.genfromtxt:

```
In [x]: np.genfromtxt(fname='data.txt',delimiter=[2,1,9,3],
                      dtype='i4, i4, f8, f8')
array([(1, 2, 100.231, 0.03), (1, 1, 1201.842, 0.04), (1, 1, 99.324, 0.02)],
      dtype=[('f0', '<i4'), ('f1', '<i4'), ('f2', '<f8'), ('f3', '<f8')])
```

as required.

Missing data

If a data set is incomplete, np.loadtxt will be unable to parse the fields with missing data into valid values for the array and will raise an exception. np.genfromtxt, however, sets missing or invalid entries equal to the default values given in Table 6.5.

For example, the comma-separated file here has two ways of indicating missing data: empty fields and entries with '???':

Table 6.5 Default filling values for missing data used by `genfromtxt`

Data type	Default value
`int`	`-1`
`float`	`np.nan`
`bool`	`False`
`complex`	`np.nan + 0.j`

```
10.1,4,-0.1,2
10.2,4,,0
10.3,???,,4
10.4,2,0.,
10.5,-1,???,3
```

Accordingly, `np.genfromtxt` sets the missing fields to its defaults:

```
In [x]: data = np.genfromtxt(fname='data.txt', dtype='f8, i4, f8, i4',
   ...:                      delimiter=',')
In [x]: print(data)
[(10.1, 4, -0.1, 2) (10.2, 4, nan, 0) (10.3, -1, nan, 4) (10.4, 2, 0.0, -1)
 (10.5, -1, nan, 3)]
```

The `missing_values` and `filling_values` arguments allow closer control over which default values to use for which columns. If `missing_values` is given as a sequence of strings, each string is associated with a column in the data file, in order; if given as a dictionary of string values, the keys denote either column indexes (if they are integers) or column names (if they are strings). The corresponding argument, `filling_values`, maps these column indexes or names to default values. If `filling_values` is provided as a single value, this value is used for missing data in all columns.

For example, to replace the invalid values in column 1 (indicated by `'???'`) with 999, the missing or invalid values in column 2 (also indicated by `'???'`) with −99 and the missing values in column 3 with 0:

```
In [x]: data =np.genfromtxt(fname='data.txt', dtype='f8, i4, f8, i4',
   ...:                    delimiter=',', missing_values={1: '???', 2: '???'},
   ...:                    filling_values={1: 999, 2: -99., 3: 0})
   ...:
In [x]: print(data)
[(10.1, 4, -0.1, 2) (10.2, 4, -99.0, 0) (10.3, 999, -99.0, 4)
 (10.4, 2, 0.0, 0) (10.5, -1, -99.0, 3)]
```

Note in particular how the missing entry in the second column has been replaced by `999` instead of the default `-1` – this would be particularly important if `-1` is a valid value for this column (however, it is now up to the rest of your code to recognize and know what to do with values such as `999`.[15]

[15] For more advanced handling of missing values, see the `genfromtxt` documentation for details on the `usemask` argument and *masked arrays* in general.

Column names

The argument `names` provides a way of setting names for the columns of data read in. If it is the boolean value `True`, the names are read from the first valid line after the number of lines skipped over specified by the `skip_header` argument; if `names` is a comma-separated string of names or a sequence of strings, those strings will be used as names. By default, `names` is `None` and the field names are taken from the `dtype`, if given.

Example E6.7 In an experiment to investigate the *Stroop effect*, a group of students were timed reading out 25 randomly ordered color names, first in black ink and then in a color other than the one they name (e.g., the word "red" in blue ink). The results are presented in the text file. Missing data are indicated by the character `X`.

```
Subject Number, Gender, Time (words in black), Time (words in color)
1,F,18.72,31.11
2,F,21.14,52.47
3,F,19.38,33.92
4,M,22.03,50.57
5,M,21.41,29.63
6,M,15.18,24.86
7,F,14.13,33.63
8,F,19.91,42.39
9,F,X,43.60
10,F,26.56,42.31
11,F,19.73,49.36
12,M,18.47,31.67
13,M,21.38,47.28
14,M,26.05,45.07
15,F,X,X
16,F,15.77,38.36
17,F,15.38,33.07
18,M,17.06,37.94
19,M,19.53,X
20,M,23.29,49.60
21,M,21.30,45.56
22,M,17.12,42.99
23,F,21.85,51.40
24,M,18.15,36.95
25,M,33.21,61.59
```

We can read in this data with `np.genfromtxt` and summarize the results with the code here.

Listing 6.5 Analyzing data from a Stroop effect experiment

```
# eg6-stroop.py
import numpy as np

# Read in the data from stroop.txt, identifying missing values and
# replacing them with NaN
❶ data = np.genfromtxt('stroop.txt', skip_header=1,
                     dtype=[('student','u8'), ('gender','S1'),
                            ('black','f8'), ('color','f8')],
                     delimiter=',',
                     missing_values='X')
```

```
nwords = 25

# Remove invalid rows from data set
❷ filtered_data = data[np.isfinite(data['black']) & np.isfinite(data['color'])]

# Extract rows by gender (M/F) and word color (black/color) and normalize
# to time taken per word
fb = filtered_data['black'][filtered_data['gender']==b'F'] / nwords
mb = filtered_data['black'][filtered_data['gender']==b'M']  / nwords
fc = filtered_data['color'][filtered_data['gender']==b'F'] / nwords
mc = filtered_data['color'][filtered_data['gender']==b'M'] / nwords

# Produce statistics: mean and standard deviation by gender and word color
mu_fb, sig_fb = np.mean(fb), np.std(fb)
mu_fc, sig_fc = np.mean(fc), np.std(fc)
mu_mb, sig_mb = np.mean(mb), np.std(mb)
mu_mc, sig_mc = np.mean(mc), np.std(mc)

print('Mean and (standard deviation) times per word (sec)')
print('gender |    black      |     color     | difference')
print('   F   | {:4.3f} ({:4.3f}) | {:4.3f} ({:4.3f}) |    {:4.3f}'
                    .format(mu_fb, sig_fb, mu_fc, sig_fc, mu_fc - mu_fb))
print('   M   | {:4.3f} ({:4.3f}) | {:4.3f} ({:4.3f}) |    {:4.3f}'
                    .format(mu_mb, sig_mb, mu_mc, sig_mc, mu_mc - mu_mb))
```

❶ In the absence of any provided `filling_values`, `np.genfromtxt` will replace the invalid fields with `np.nan`.

❷ We only want to consider students with times for both parts of the experiment, so create a filtered data set here.

The output shows a significantly slower per-word speed for the false-colored words than for the words in black:

```
Mean and (standard deviation) times per word (sec)
gender |    black      |     color     | difference
   F   | 0.770 (0.137) | 1.632 (0.306) |    0.862
   M   | 0.849 (0.186) | 1.679 (0.394) |    0.830
```

6.2.4 Exercises

Problems

P6.2.1 The following text file gives some data concerning the 8,000 m peaks, in alphabetical order.

```
ex6-2-b-mountain-data.txt This file contains a list of the 14
highest mountains in the world with their names, height, year
of first ascent, year of first winter ascent, and location as
longitude and latitude in degrees (d), minutes (m) and seconds
(s). Note: as of 2013, no winter ascent has been made of K2 or
Nanga Parbat.
```

```
----------------------------------------------------------------------
Name         Height  First ascent First winter Location
               m        date      ascent date   (WGS84)
----------------------------------------------------------------------
Annapurna I   8091    3/6/1950    3/2/1987     28d35m46sN 83d49m13sE
Broad Peak    8051    9/6/1957    5/3/2013     35d48m39sN 76d34m06sE
Cho Oyu       8201    19/10/1954  12/2/1985    28d05m39sN 86d39m39sE
Dhaulagiri I  8167    13/5/1960   21/1/1985    27d59m17sN 86d55m31sE
Everest       8848    29/5/1953   17/2/1980    27d59m17sN 86d55m31sE
Gasherbrum I  8080    5/7/1958    9/3/2012     35d43m28sN 76d41m47sE
Gasherbrum II 8034    7/7/1956    2/2/2011     35d45m30sN 76d39m12sE
K2            8611    31/7/1954   -            35d52m57sN 76d30m48sE
Kangchenjunga 8568    25/5/1955   11/1/1986    27d42m09sN 88d08m54sE
Lhotse        8516    18/5/1956   31/12/1988   27d57m42sN 86d56m00sE
Makalu        8485    15/5/1955   9/2/2009     27d53m21sN 87d05m19sE
Manaslu       8163    9/5/1956    12/1/1984    28d33m0sN  84d33m35sE
Nanga Parbat  8126    3/7/1953    -            35d14m15sN 74d35m21sE
Shishapangma  8027    2/5/1964    14/1/2005    28d21m8sN  85d46m47sE
----------------------------------------------------------------------
```

Use NumPy's `loadtxt` method to read these data into a suitable structured array to determine the following:

1. The lowest 8,000 m peak
2. The most northely, easterly, southerly and westerly peaks
3. The most recent first ascent of the peaks
4. The first of the peaks to be climbed in winter

Also produce another structured array containing a list of mountains with their height in *feet* and first ascent date, ordered by increasing height.[16]

P6.2.2 The file `busiest_airports.txt`, available to download from scipython.com/ex/afa, provides details of the 30 busiest airports in the world in 2014. The tab-delimited fields are: three-letter IATA code, airport name, airport location, latitude and longitude (both in degrees).

Write a program to determine the distance between two airports identified by their three-letter IATA code, using the Haversine formula (see, for example, Exercise 4.4.2) and assuming a spherical Earth of radius 6378.1 km.

P6.2.3 The World Bank provides an extensive collection of data sets on a wide range of "indicators," which is searchable at http://data.worldbank.org/. Data sets concerning child immunization rates for BCG (against tuberculosis), Pol3 (Polio) and measles in three South-East Asian countries between 1960 and 2013 are available at scipython.com/ex/afb. Fields are delimited by semicolons and missing values are indicated by `'..'`.

Use NumPy methods to read in this data and create three plots (one for each vaccine) comparing immunization rates in the three countries.

[16] 1 metre = 3.2808399 feet.

6.3 Statistical methods

NumPy provides several methods for performing statistical analysis, either on an entire array or an axis of it.

6.3.1 Ordering statistics

Maxima and minima

We have already used `np.min` and `np.max` to find the minimum and maximum values of an array (these methods are also available using the names `np.amin` and `np.amax`). If the array contains one or more NaN values, the corresponding minimum or maximum value will be `np.nan`. To ignore NaN values instead, use `np.nanmin` and `np.nanmax`:

```
In [x]: a = np.sqrt(np.linspace(-2, 2, 4))
In [x]: print(a)
[        nan         nan  0.          1.          1.41421356]
In [x]: np.min(a), np.max(a)
Out[x]: (nan, nan)
In [x]: np.nanmin(a), np.nanmax(a)
(0.0, 1.4142135623730951)
```

We have also met the functions `np.argmin` and `np.argmax`, which return the *index* of the minimum and maximum values in an array; they too have `np.nanargmin` and `np.nanargmax` variants:

```
In [x]: np.argmin(a), np.argmax(a)
Out[x]: (0, 0)          # The first nan in the array
In [x]: np.nanargmin(a), np.nanargmax(a)
Out[x]: (2, 4)          # The indexes of 0, 1.41421356
```

The related methods, `np.fmin` / `np.fmax` and `np.minimum` / `np.maximum`, compare two arrays, *element by element* and return another array of the same shape. The first pair of methods ignores NaN values and the second pair propagates them into the output array. For example,

```
In [x]: np.fmin([1, -5, 6, 2], [0, np.nan, -1, -1])
array([ 0., -5., -1., -1.])         # NaNs are ignored
In [x]: np.maximum([1, -5, 6, 2], [0, np.nan, -1, -1])
array([  1.,  nan,   6.,   2.])     # NaNs are propagated
```

Percentiles

The `np.percentile` method returns a specified percentile, `q`, of the data along an axis (or along a flattened version of the array if no axis is given). The minimum of an array is the value at `q=0` (0th percentile), the maximum is the value at `q=100` (100th percentile) and the median is the value at `q=50` (50th percentile). Where no single value in the array corresponds to the requested value of `q` exactly, a weighted average of the two nearest values is used. For example,

```
In [x]:  a = np.array([[0., 0.6, 1.2], [1.8, 2.4, 3.0]])
In [x]: np.percentile(a, 50)
1.5
```

```
In [x]: np.percentile(a, 75)
2.25
In [x]: np.percentile(a, 50, axis=1)
array([ 0.6,  2.4])
In [x]: np.percentile(a, 75, axis=1)
array([ 0.9,  2.7])
```

6.3.2 Averages, variances and correlations

Averages

In addition to `np.mean`, which calculates the arithmetic mean of the values along a specified axis of an array, NumPy provides methods for calculating the weighted average, median, standard deviation and variance. The weighted average is calculated as

$$\bar{x}_w = \frac{\sum_i^N w_i x_i}{\sum_i^N w_i}$$

where the weights, w_i, are supplied as a sequence the same length as the array. For example,

```
In [x]: x = np.array([1., 4., 9., 16.])
In [x]: np.mean(x)
7.5
In [x]: np.median(x)
6.5
In [x]: np.average(x, weights=[0., 3., 1., 0.])
5.25        # ie (3.*4. + 1.*9.) / (3. + 1.)
```

If you want the sum of the weights as well as the weighted average, set the `returned` argument to `True`. In the following example, we do this and find the weighted averages in each row (`axis=1` averages values across *columns* of a two-dimensional array):

```
In [x]: x = np.array( [[1., 8., 27], [-0.5, 1., 0.]] )
In [x]: av, sw = np.average(x, weights=[0., 1., 0.1], axis=1, returned=True)
In [x]: print(av)
[ 9.72727273  0.90909091]
In [x]: print(sw)
[ 1.1  1.1]
```

The averages are therefore $(1 \times 8 + 0.1 \times 27)/1.1 = 9.72727273$ and $(1 \times 1.)/1.1 = 0.90909091$ where 1.1 is the sum of the weights.

Standard deviations and variances

The function `np.std` calculates, by default, the *uncorrected sample standard deviation*:

$$\sigma_N = \sqrt{\frac{1}{N}\sum_i^N (x_i - \bar{x})^2}.$$

where x_i are the N observed values in the array and $\bar{x}$ is their mean. To calculate the *corrected sample standard deviation*,

$$\sigma = \sqrt{\frac{1}{N-\delta}\sum_i^N (x_i - \bar{x})^2},$$

pass to the argument `ddof` the value of δ such that $N - \delta$ is the number of degrees of freedom in the sample. For example, if the sample values are drawn from the population independently with replacement and used to calculate $\bar{x}$ there are $N - 1$ degrees of freedom in the vector of residuals used to calculate σ: $(x_1 - \bar{x}, x_2 - \bar{x}, \cdots, x_N - \bar{x})$ and so $\delta = 1$. For example,

```
In [x]: x = np.array([1., 2., 3., 4.])
In [x]: np.std(x)          # or x.std(), uncorrected standard deviation
1.1180339887498949
In [x]: np.std(x, ddof=1)   # corrected standard deviation
1.2909944487358056
```

The function `np.nanstd` calculates the standard deviation ignoring `np.nan` values (so that N is the number of non-NaN values in the array). NumPy also has methods for calculating the *variance* of the values in an array: `np.var` and `np.nanvar`.

The covariance is returned by the `np.cov` method. In its simplest invocation, it can be passed a single two-dimensional array, `X`, in which the rows represent variables, x_i, and the columns observations of the value of each variable. `np.cov(X)` then returns the covariance matrix, C_{ij}, indicating how variable x_i varies with x_j: the element C_{ij} is said to be an estimate of the covariance of variables x_i and x_j:

$$C_{ij} \equiv \mathrm{cov}(x_i, x_j) = \mathrm{E}[(x_i - \mu_i)(x_j - \mu_j)]$$

where μ_i is the mean of the variable x_i and E[ ] denotes the expected value. If there are N observed values for each of the variables, $\mu_i = \frac{1}{N}\sum_k x_{ik}$. The *unbiased* estimate of the covariance is then

$$C_{ij} = \frac{1}{N-1}\sum_k [(x_{ik} - \mu_i)(x_{jk} - \mu_j)]$$

This is the default behavior of `np.cov`, but if the `bias` argument is set to 1, then N is used in the denominator here to give the *biased* estimate of the covariance. Finally, the denominator can be set explicitly to $N - \delta$ by passing δ as the argument to the `ddof` argument of `cov`.

Example E6.8 As an example, consider the matrix of five observations each of three variables, x_0, x_1 and x_2 whose observed values are held in the three rows of the array `X`:

```
X = np.array([ [0.1, 0.3, 0.4, 0.8, 0.9],
               [3.2, 2.4, 2.4, 0.1, 5.5],
               [10., 8.2, 4.3, 2.6, 0.9]
             ])
```

The covariance matrix is a 3×3 array of values,

```
In [x]: print( np.cov(X) )
[[  0.115 ,   0.0575,  -1.2325],
 [  0.0575,   3.757 ,  -0.8775],
 [ -1.2325,  -0.8775,  14.525 ]]
```

The diagonal elements, C_{ii}, are the variances in the variables x_i assuming $N-1$ degrees of freedom:

```
In [x]: print(np.var(X, axis=1, ddof=1))
[  0.115   3.757  14.525]
```

Although the magnitude of the covariance matrix elements is not always easy to interpret (because it depends on the magnitude of the individual observations which may be very different for different variables), it is clear that there is a strong anticorrelation between x_0 and x_2 ($C_{02} = -1.2325$: as one increases the other decreases) and no strong correlation between x_0 and x_1 ($C_{01} = 0.0575$: x_0 and x_1 do not trend strongly together).

The *correlation coefficient matrix* is often used in preference to the covariance matrix as it is normalized by dividing C_{ij} by the product of the variables' standard deviations:

$$P_{ij} = \mathrm{corr}(x_i, x_j) = \frac{C_{ij}}{\sigma_i \sigma_j} = \frac{C_{ij}}{\sqrt{C_{ii}C_{jj}}}.$$

This means that the elements P_{ij} have values between -1 and 1 inclusive, and the diagonal elements, $P_{ii} = 1$. In our example, using `np.corrcoef` gives:

```
In [x]: print( np.corrcoef(X) )
[[ 1.          0.0874779  -0.95363007]
 [ 0.0874779   1.         -0.11878687]
 [-0.95363007 -0.11878687  1.        ]]
```

It is easy to see from this correlation coefficient matrix the strong anticorrelation between x_0 and x_2 ($C_{0,2} = -0.954$) and the lack of correlation between x_1 and the other variables (e.g., $C_{1,0} = 0.087$).

Both the `np.cov` and `np.corrcoef` methods can take a second array-like object containing a further set of variables and observations, so they can be called on a pair of one-dimensional arrays without stacking them into a single matrix:

```
In [x]: x = np.array([1., 2., 3., 4., 5.])
In [x]: y = np.array([0.08, 0.31, 0.41, 0.48, 0.62])
In [x]: print( np.corrcoef(x,y) )
[[ 1.          0.97787645]
 [ 0.97787645  1.        ]]
```

That is

```
np.corrcoef(x, y)
```

is a convenient alternative to

```
np.corrcoef(np.vstack((x,y)))
```

Finally, if your observations happen to be in the rows of your matrix, with the variables corresponding to the columns (instead of the other way round) there is no need

to transpose the matrix, just pass `rowvar=0` to either `np.cov` or `np.corrcoef` and NumPy will take care of it for you.

Example E6.9 The Cambridge University Digital Technology Group have been recording the weather from the roof of their department building since 1995 and make the data available to download in a single CSV file at www.cl.cam.ac.uk/research/dtg/weather/.

The following program determines the correlation coefficient between pressure and temperature at this site.

Listing 6.6 Calculating the correlation coefficient between air temperature and pressure

```
# eg6-pT.py
import numpy as np
import pylab

data = np.genfromtxt('weather-raw.csv', delimiter=',', usecols=(1,4))
# Remove any rows with either missing T or missing p
data = data[~np.any(np.isnan(data), axis=1)]
# Temperatures are reported after multiplication by a factor of 10 so remove
# this factor
data[:,0] /= 10

# Get the correlation coefficient
corr = np.corrcoef(data, rowvar=0)[0,1]
print('p-T correlation coefficient: {:.4f}'.format(corr))

# Plot the data on a scatterplot: T on x-axis, p on y-axis.
pylab.scatter(*data.T, marker='.')
pylab.xlabel('$T$ /$\mathrm{^\circ C}$')
pylab.ylabel('$p$ /mbar')
pylab.show()
```

The output (Figure 6.4) gives a correlation coefficient of 0.0260: as expected, there is little correlation between air temperature and pressure (since the air density also varies).

6.3.3 Histograms

The NumPy function, `np.histogram`, creates a histogram from the values in an array. That is, a set of *bins* is defined with lower and upper limits and each is filled with the number of elements from the array whose value falls within its limits. For example, suppose the following array holds the percentage marks of 10 students in a test:

```
In [x]: marks = np.array([45, 68, 56, 23, 60, 87, 75, 59, 63, 72])
```

There are several ways to define the histogram bins. If the `bins` argument is a sequence, it defines the boundaries of the sequential bins:

```
In [x]: bins = [20, 40, 60, 80, 100]
```

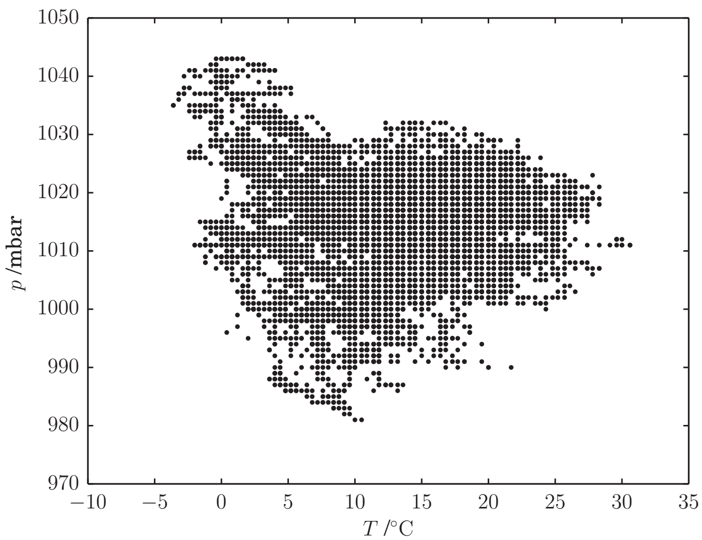


Figure 6.4 There is virtually no correlation between air temperature and air pressure in this data set.

defines four bins with ranges [20 – 40%), [40 – 60%), [60 – 80%) and [80 – 100%]. All but the last bin is half open; that is, the first bin includes marks from and including 20% up to but not including 40%. Note that a sequence of $N + 1$ numbers is required to create N bins. The `np.histogram` method returns a tuple consisting of the values of the histogram and the bin edges we defined (both as NumPy arrays).

```
In [x]: hist, bins = np.histogram(marks, bins)
In [x]: hist
Out[x]: array([1, 3, 5, 1])

In [x]: bins
Out[x]: array([ 20,  40,  60,  80, 100])
```

This shows that there is one mark in the 20 – 40% bin, three in the 40 – 60% bin and so on.

If you just want a certain number of evenly spaced bins, an integer can be passed as `bins` instead of a sequence:

```
In [x]: np.histogram(marks, bins=5)
Out[x]: (array([1, 1, 2, 4, 2]),
         array([ 23. ,  35.8,  48.6,  61.4,  74.2,  87. ]))
```

By default, the requested number of bins range between the minimum and maximum values of the array (here, `23` and `87`); to specify a different minimum and maximum, set the `range` argument tuple:

```
In [x]: np.histogram(marks, bins=5, range=(0,100))
Out[x]: (array([0, 1, 3, 5, 1]),
         array([   0.,   20.,   40.,   60.,   80.,  100.]))
```

The `np.histogram` method also has an optional boolean argument `density`: by default it is `False`, meaning that the histogram array returned contains the *number* of

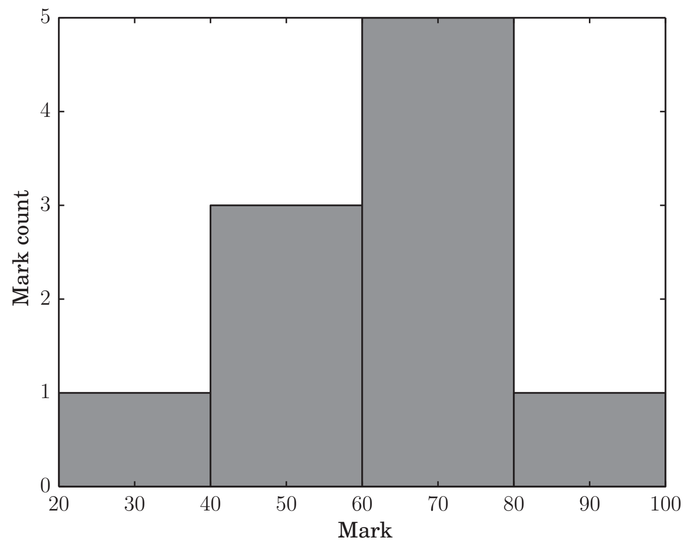


Figure 6.5 An example histogram.

values from the original array in each bin. If `density` is set to `True`, the histogram array will contain the *probability density function*, normalized so that the integral over the entire range of the bins is equal to unity:

```
In [x]: hist, bins = np.histogram(marks, bins=5, range=(0,100),
                                  density=True)
In [x]: print(hist)
[ 0.      0.005   0.015   0.025   0.005]
In [x]: bin_width = 100/5
In [x]: print(np.sum(hist) * bin_width)
1.0
```

(By integral here we mean the area under the histogram, which is the sum of each histogram bar height times its corresponding bin width.)

To plot a histogram with `pylab`, use `pylab.hist`, passing it the same arguments you would to `np.histogram`:[17]

```
❶ In [x]: hist, bins, patches = pylab.hist(marks, bins=5, range=(0,100))
In [x]: hist, bins
Out[x]:
(array([ 0.,   1.,   3.,   5.,   1.]),
 array([   0.,    20.,    40.,    60.,    80.,   100.]))
In [x]: pylab.show()
```

❶ In addition to the bin counts (`hist`) and boundaries (`bins`), `pylab` returns a list of references to the "patches" which appear in the plotted figure (see Section 7.1.5 for more information about this advanced feature).

The resulting histogram is plotted in Figure 6.5. See also Sections 3.3.2 and 7.1.2.

[17] Note that the `density` argument is not supported as of Matplotlib 1.3.1: instead, set `normed=True` for a probability density plot.

6.3.4 Exercises

Problems

P6.3.1 A certain lottery involves players selecting six numbers without replacement from the range [1,49]. The jackpot is shared among the players who match all six numbers ("balls") selected in the same way at random in a twice-weekly draw (in any order). If no player matches every drawn number, the jackpot "rolls over" and is added to the following draw's jackpot.

Although the lottery is *fair* in the sense that every combination of drawn numbers is equally likely, it has been observed that many players show a preference in their selection for certain numbers, such as those that represent dates (i.e., more of their numbers are chosen from [1,31] than would be expected if they chose randomly). Hence, to avoid sharing the jackpot and hence to maximize one's expected winnings, it would be reasonable to avoid these numbers.

Test this hypothesis by establishing if there is any correlation between the number of balls with values less than 13 (representing a month) and the jackpot winnings per person. Ignore draws immediately following a rollover. The necessary data can be downloaded from scipython.com/ex/afe.

P6.3.2 We have seen how to create a histogram plot from an array with `pylab.hist`, but suppose you have already created arrays `hist` and `bins` using `np.hist` and want to plot the resulting histogram from these arrays. You can't use `pylab.hist` because this function expects to act on the original array of data. Use `pylab.bar`[18] to plot a `hist` array as a bar chart.

P6.3.3 The heights, in cm, of a sample of 1,000 adult men and 1,000 adult women from a certain population are collected in the data files `ex6-3-f-male-heights.txt` and `ex6-3-f-female-heights.txt` available at scipython.com/ex/afd. Read in the data and establish the mean and standard deviation for each sex. Create histograms for the two data sets using a suitable binning interval and plot them on the same figure.

Repeat the exercise in imperial units (feet and inches).

6.4 Polynomials

NumPy provides a powerful set of classes for representing polynomials, including methods for evaluation, algebra, root-finding and fitting of several kinds of polynomial basis functions. In this section, the simplest and most familiar basis, the power series, will be described first, before a discussion of a few other classical orthogonal polynomial basis functions.

[18] Documentation for this method is at http://matplotlib.org/api/pyplot_api.html/matplotlib.pyplot.bar; see also Section 7.1.2.

6.4.1 Defining and evaluating a polynomial

A (finite) polynomial power series has as its basis the powers of x: $1(= x^0), x, x^2, x^3, \cdots, x^N$, with coefficients c_i:

$$P(x) = \sum_{i=0}^{N} c_0 + c_1 x + c_2 x^2 + c_3 x^3 + \cdots + c_N x^N$$

This section describes the use of the `Polynomial` *convenience class* which provides a natural interface to the underlying functionality of NumPy's `polynomial` package.

The polynomial convenience class is `numpy.polynomial.Polynomial`. To import it directly, use

```
In [x]: from numpy.polynomial import Polynomial
```

Alternatively, if the whole NumPy library is already imported as `np`, then rather than constantly refer to this class as `np.polynomial.Polynomial`, it is convenient to define a variable:

```
In [x]: import numpy as np
In [x]: Polynomial = np.polynomial.Polynomial
```

This is the way we will refer to the `Polynomial` class in this book.

To define a polynomial object, pass the `Polynomial` constructor a sequence of coefficients to increasing powers of x, starting with c_0. For example, to represent the polynomial

$$P(x) = 6 - 5x + x^2$$

define a the object

```
In [x]: p = Polynomial([6, -5, 1])
```

You can inspect the coefficients of a `Polynomial` object with `print` or by referring to its `coef` attribute.

```
In [x]: print(p)
poly([ 6. -5.  1.])
In [x]; P.coef
Out[x]: array([ 6., -5.,  1.])
```

Notice that the integer coefficients used to define the polynomial have been automatically cast to `float`. It is also possible to use `complex` coefficients.

To evaluate a polynomial for a given value of x, "call" it as follows:

```
In [x]: p(4)    # calculate p at a single value of x
2.0
In [x]: x = np.linspace(-5, 5, 11)
In [x]: print(p(x))     # calculate p on a sequence of x values
Out[x]: [ 56.  42.  30.  20.  12.   6.   2.   0.   0.   2.   6.]
```

6.4.2 Polynomial algebra

The `Polynomial` convenience class implements the familiar Python operators: `+`, `-`, `*`, `//`, `**`, `%` and `divmod`[19] on `Polynomial` objects. These are illustrated in the following examples using the polynomials

$$P(x) = 6 - 5x + x^2$$
$$Q(x) = 2 - 3x$$

```
In [x]: p = Polynomial([6, -5, 1])
In [x]: q = Polynomial([2, -3])
In [x]: print(p + q)
poly([ 8. -8.  1.])

In [x]: print(p - q)
poly([ 4. -2.  1.])

In [x]: print(p * q)
poly([ 12. -28.  17.  -3.])

In [x]: print(p // q)
poly([ 1.44444444 -0.33333333])

In [x]: print(p % q)
poly([ 3.11111111])     # i.e. 28/9
```

Division of a polynomial by another polynomial is analogous to integer division (and uses the same `//` operator): that is, the result is another polynomial (with no reciprocal powers of x), possibly leaving a remainder.

Hence $p = q(-\frac{1}{3}x + \frac{13}{9}) + \frac{28}{9}$ and the `//` operator returns the quotient polynomial, $-\frac{1}{3}x + \frac{13}{9}$. The remainder (which, in general, will be another polynomial) is returned, as might be expected, by the modulus operator, `%`. The `divmod()` built-in returns both quotient and remainder in a tuple:

```
In [x]: quotient, remainder = divmod(p, q)
In [x]: print(quotient)

poly([ 1.44444444 -0.33333333])     # i.e. p(x) // q(x) is 13/9 - x/3

In [x]: print(remainder)
poly([ 3.11111111])
```

Exponentiation is supported through the `**` operator; polynomials can only be raised to a non-negative integer power:

```
In [x]: print(q ** 2)
poly([  4. -12.   9.])
```

It isn't always convenient to create a new polynomial object in order to use these operators on one another, so many of the operators described here also work with scalars:

[19] The `divmod` function returns the quotient and remainder of a division operation as a tuple.

```
In [x]: print(p * 2)     # multiplication by a scalar
poly([ 12. -10.   2.])
In [x]: print(p / 2)     # division by a scalar
poly([ 3.  -2.5  0.5])
```

and even tuples, lists and arrays of polynomial coefficients. For example, to multiply $P(x)$ by $x^2 - 2x^3$:

```
In [x]: print(p * [0, 0, 1, -2])
poly([  0.   0.   6. -17.  11.  -2.])
```

Finally, one polynomial can be substituted into another. To evaluate $P(Q(x))$, simply use `p(q)`:

```
In [x]: print(p(q))
poly([ 0.  3.  9.])
```

That is, $P(Q(x)) = 3x + 9x^2$.

6.4.3 Root-finding

The roots of a polynomial are returned by the `roots` method. Repeated roots are simply repeated in the returned array:

```
In [x]: p.roots()
array([ 2.,  3.])
In [x]: (q*q).roots()
array([ 0.66666667,  0.66666667])
In [x]: Polynomial([5, 4, 1]).roots()
array([-2.-1.j, -2.+1.j])
```

Polynomials can also be created from their roots with `Polynomial.fromroots`:

```
In [x]: print( Polynomial.fromroots([-4, 2, 1]) )
poly([  8. -10.   1.   1.])
```

That is, $(x + 4)(x - 2)(x - 1) = 8 - 10x + x^2 + x^3$. Note that the way the polynomial is constructed means that the coefficient of the highest power of x will be 1.

Example E6.10 The tanks used in the storage of cryogenic liquids and rocket fuel are often spherical (why?). Suppose a particular spherical tank has a radius R and is filled with a liquid to a height h. It is (fairly) easy to find a formula for the volume of liquid from the height:

$$V = \pi Rh^2 - \frac{1}{3}\pi h^3.$$

Suppose that there is a *constant* flow of liquid from the tank at a rate $F = -\mathrm{d}V/\mathrm{d}t$. How does the height of liquid, h, vary with time? Differentiating the earlier mentioned equation with respect to t leads to

$$(2\pi Rh - \pi h^2)\frac{\mathrm{d}h}{\mathrm{d}t} = -F.$$

If we start with a full tank ($h = 2R$) at time $t = 0$, this ordinary differential equation may be integrated to yield the equation

$$-\frac{1}{3}\pi h^3 + \pi R h^2 + \left(Ft - \frac{4}{3}\pi R^3\right) = 0,$$

a cubic polynomial in h. Because this equation cannot be inverted analytically for h, let's use NumPy's `Polynomial` class to find $h(t)$, given a tank of radius $R = 1.5$ m from which liquid is being drawn at 200 cm^3 s^{-1}.

The total volume of liquid in the full tank is $V_0 = \frac{4}{3}\pi R^3$. Clearly, the tank is empty when $h = 0$, which occurs at time $T = V_0/F$, since the flow rate is constant. At any particular time, t, we can find h by finding the roots of this equation.

Listing 6.7 Liquid height in a spherical tank

```
# eg6-c-spherical-tank-a.py
import numpy as np
import pylab
Polynomial  = np.polynomial.Polynomial

# Radius of the spherical tank in m
R = 1.5
# Flow rate out of the tank, m^3.s-1
F = 2.e-4
# Total volume of the tank
V0 = 4/3 * np.pi * R**3
# Total time taken for the tank to empty
T = V0 / F

# coefficients of the quadratic and cubic terms
# of p(h), the polynomial to be solved for h
c2, c3 = np.pi * R, -np.pi / 3

N = 100
# array of N time points between 0 and T inclusive
time = np.linspace(0, T, N)                                   ❶
# create the corresponding array of heights h(t)
h = np.zeros(N)
for i, t in enumerate(time):
    c0 = F*t - V0
    p = Polynomial([c0, 0, c2, c3])
    # find the three roots to this polynomial
    roots = p.roots()                                         ❷
    # we want the one root for which 0 <= h <= 2R
    h[i] = roots[(0 <= roots) & (roots <= 2*R)][0]

pylab.plot(time, h, 'o')
pylab.xlabel('Time /s')
pylab.ylabel('Height in tank /m')
pylab.show()
```

❶ We construct an array of time points between $t = 0$ and $t = T$.

❷ For each time point find the roots of the above cubic polynomial. Only one of the roots is physically meaningful, in that $0 \leq h \leq 2R$ (the height of the level of liquid

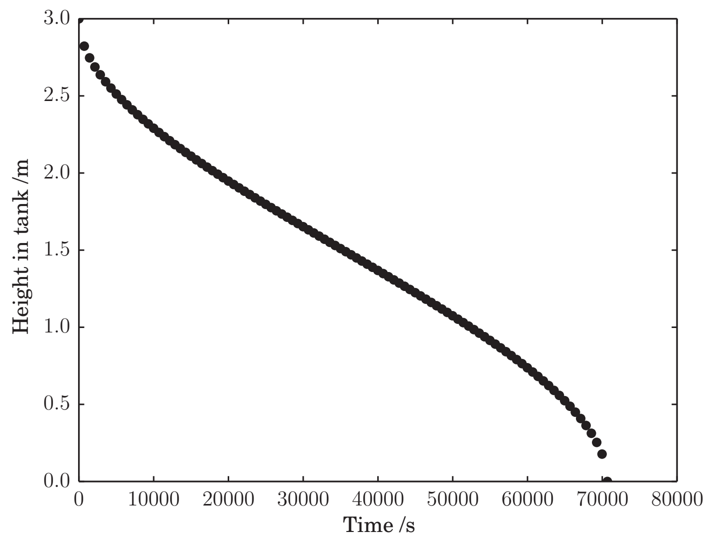


Figure 6.6 The height of liquid as a function of time, $h(t)$, for the spherical tank problem.

cannot be negative or greater than the diameter of the tank), so we extract that root (by boolean indexing) and store it in the array h.

Finally, we plot h as a function of time (Figure 6.6).

6.4.4 Calculus

Polynomials can be differentiated with the `Polynomial.deriv` method. By default, this function returns the first derivative, but the optional argument `m` can be set to return the mth derivative:

```
In [x]: print(p)
poly([ 6. -5.  1.]) # 6 - 5x + x^2
In [x]: print(p.deriv())
poly([-5.  2.])
In [x]: print(p.deriv(2))
poly([ 2.])
```

A `Polynomial` object can also be integrated with an optional lower bound, L, and constant of integration, k, treated as shown in the following example:

$$\int_L^x 2 - 3x\,\mathrm{d}x = \left[2x - \tfrac{3}{2}x^2\right]_L^x = 2x - \tfrac{3}{2}x^2 - 2L + \tfrac{3}{2}L^2$$

$$\int 2 - 3x\,\mathrm{d}x = 2x - \tfrac{3}{2}x^2 + k$$

By default, L and k are zero, but can be specified by passing the arguments `lbnd` and `k` to the `Polynomial.integ` method:

```
In [x]: print(q)
poly([ 2. -3.])
In [x]: print(q.integ())
poly([ 0.   2.  -1.5])
```

```
In [x]: print(q.integ(lbnd=1))
poly([-0.5  2.  -1.5])
In [x]: print(q.integ(k=2))
poly([ 2.   2.  -1.5])
```

Polynomials can be integrated repeatedly by passing a value to `m`, giving the number of integrations to perform.[20]

6.4.5 ◊ Classical orthogonal polynomials

In addition to the `Polynomial` class representing simple power series such as $a_0 + a_1x + a_2x^2 + \cdots + a_nx^n$, NumPy provides classes to represent a series composed of any of a number of classical orthogonal polynomials. These polynomials and linear combinations of them are widely used in physics, statistics and mathematics. As of NumPy version 1.8, the polynomial convenience classes provided are `Chebyshev`, `Legendre`, `Laguerre`, `Hermite` ("physicists' version") and `HertmiteE` ("probabilists' version"). Many good textbooks exist describing the properties of these polynomial classes; to illustrate their use we will focus here on the Legendre polynomials,[21] denoted $P_n(x)$. These are the solutions to Legendre's differential equation,

$$\frac{\mathrm{d}}{\mathrm{d}x}\left[(1-x^2)\frac{\mathrm{d}}{\mathrm{d}x}P_n(x)\right] + n(n+1)P_n(x) = 0.$$

The first few Legendre polynomials are

$$\begin{aligned}
P_0(x) &= 1\\
P_1(x) &= x\\
P_2(x) &= \tfrac{1}{2}(3x^2 - 1)\\
P_3(x) &= \tfrac{1}{2}(5x^3 - 3x)\\
P_4(x) &= \tfrac{1}{8}(35x^4 - 30x^2 + 3)
\end{aligned}$$

and are plotted in Figure 6.7.

A useful property of the Legendre polynomials is their orthogonality on the interval [-1, 1]:

$$\int_{-1}^{1} P_n(x)P_m(x)\,\mathrm{d}x = \frac{2}{2n+1}\delta_{mn}$$

which is important in their use as a basis for representing suitable functions.[22]

To create a linear combination of Legendre polynomials, pass the coefficients to the `Legendre` constructor, just as for `Polynomial`. For example, to construct the polynomial expansion $5P_1(x) + 2P_2(x)$:

---

[20] Different constants of integration for each can be specified by setting `k` to an array of values.

[21] The Legendre Polynomials are named after the French mathematician Adrien-Marie Legendre (1752–1833); for 200 years until 2005 many publications mistakenly used a portrait of the unrelated French politician Louis Legendre as that of the mathematician.

[22] In particular, in physics, the multipole expansion of electrostatic potentials.

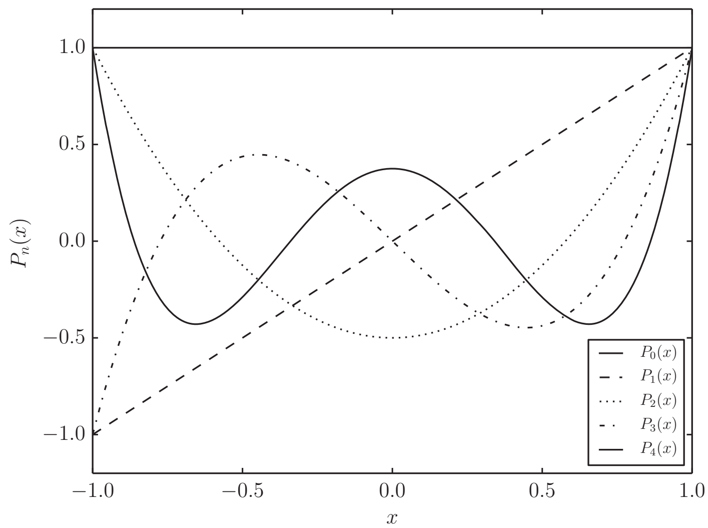


Figure 6.7 The first five Legendre Polynomials, $P_n(x)$ for $x = 0, 1, 2, 3, 4$.

```
In [x]: Legendre = np.polynomial.Legendre
In [x]: A = Legendre([0, 5, 2])
```

An existing polynomial object can be converted into a Legendre series with the `cast` method:

```
In [x]: P = Polynomial([0,1,1])
In [x]: Q = Legendre.cast(P)
In [x]: print(Q)
leg([ 0.33333333  1.          0.66666667])
```

That is, $x + x^2 = \frac{1}{3}P_0 + P_1 + \frac{2}{3}P_2$.

An instance of a single Legendre polynomial basis function can be created with the `basis` method:

```
In [x]: L3 = Legendre.basis(3)
```

creates an object representing $P_3(x)$, and is equivalent to calling `Legendre([0,0,0, 1])`. To obtain a regular power series, we can cast it back to a `Polynomial`:

```
In [x]: print(Polynomial.cast(L3))
poly([ 0.  -1.5  0.   2.5])
```

In addition to the functions just described for `Polynomial`, including differentiation and integration of polynomial series, the convenience classes for the classical orthogonal polynomials expose several useful methods.

`convert` converts between different kinds of polynomials. For example, the linear combination $A(x) = 5P_1(x) + 2P_2(x) = 5x + 2\frac{1}{2}(3x^2 - 1) = -1 + 5x + 3x^2$, as a power series of monomials (a Maclaurin series), is represented by an instance of the `Polynomial` class as:

```
In [x]: A = Legendre([0, 5, 2])
In [x]: B = A.convert(kind=Polynomial)
In [x]: print(B)
In [x]: poly([-1.  5.  3.])
```

Because the objects `A` and `B` represent the same underlying function (just expanded in different basis sets) they evaluate to the same value when given the same x, and have the same roots:

```
In [x]: A(-2) == B(-2)
Out[x]: True
In [x]: print(A.roots(), B.roots(), sep='\n')
[-1.84712709  0.18046042]
[-1.84712709  0.18046042]
```

6.4.6 Fitting polynomials

A common use of polynomial expansions is in fitting and approximating data series. NumPy's polynomial modules provide methods for the least squares fitting of functions. The `fit` function of the polynomial convenience classes is described in this section.[23]

The `domain` and `window` attributes

A typical one-dimensional fitting problem requires the best-fit polynomial to a finite, continuous function over some finite region of the x-axis (the *domain*). However, polynomials themselves can differ from each other wildly and diverge as $x \to \pm\infty$. This makes any attempt to blindly find the least squares fit on the domain of the function itself potentially risky: the fitted polynomial is frequently subject to numerical instability, overflow, underflow and other types of ill-conditioning (see Section 9.2). As an example, consider the function

$$f(x) = \mathrm{e}^{-\sin 40x}$$

in the interval (100, 100.1). There is nothing particularly tricky about this function: it is well-behaved everywhere and $f(x)$ takes very moderate values between e^{-1} and e^{1}. Yet a straightforward least squares fit to a fourth-order polynomial on this domain gives:

$$-11.881851 + 2379.22228x - 119.741202x^2 - 23828009.7x^3 + 1192894610x^4$$

and clearly the potential for numerical instability and loss of accuracy with even moderate values of x: our approximation to $f(x)$ is built up from difference between very large monomial terms.

Each class of polynomial has a default *window* over which it is optimal to take a linear combination in fitting a function. For example, the Legendre polynomials window is the region [−1,1] plotted above, on which $P_n(x)$ are orthogonal and everywhere $|P_n(x)| < 1$. The problem is that it is rather unlikely that the function to be fitted falls

[23] *Note:* The older `np.poly1d` class representing one-dimensional polynomials is still available (as of NumPy 1.9) for backward-compatibility reasons. It is documented at http://docs.scipy.org/doc/numpy/reference/routines.polynomials.poly1d.html and provides a simpler but less reliable least squares fitting method, `polyfit`. It is recommended, however, to use the new `Polynomial` class in new code.

within the chosen polynomials' window. It is therefore necessary to relate the domain of the function to the window. This is done by shifting and scaling the x-axis: that is, by mapping points in the function's domain to points in the fitting polynomials' window. The polynomial `fit` function does this automatically, so the fourth-order least squares fit to the earlier mentioned function yields

```
In [x]: x = np.linspace(100, 100.1, 1001)
In [x]: f = lambda x: np.exp(-np.sin(40*x))
In [x]: p = Polynomial.fit(x, f(x), 4)
In [x]: print(p)
poly([ 1.49422551 -2.54641449  0.63284641  1.84246463 -1.02821956])
```

The domain and window of a polynomial can be inspected as the attributes `domain` and `window` respectively:

```
In [x]: p.domain
array([ 100. ,  100.1])
In [x]: p.window
array([-1.,  1.])
```

It is important to note that the argument `x` is mapped from the domain to the window whenever a polynomial is evaluated. This means that two polynomials with different domains and/or windows may evaluate to different values *even if they have the same coefficients*. For example, if we create a `Polynomial` object from scratch with the same coefficients as the fitted polynomial `p` above:

```
In [x]: q = Polynomial([1.49422551, -2.54641449,  0.63284641,
                        1.84246463, -1.02821956])
```

it is created with the default domain and window, which are *both* `(-1,1)`:

```
In [x]: print(q.domain, q.window)
[-1.  1.] [-1.  1.]
```

and so evaluating `q` at 100.05, say, maps 100.05 in the domain to 100.05 in the window and gives a very different answer from the evaluation of `p` at the same point in the domain (which maps to 0. in the window):

```
In [x]: q(100.05), p(100.05)
(-101176442.96772559, 1.4942255113760108)
```

It is easy to show that the mapping function from x in a domain (a, b) to x' in a window (a', b') is

$$x' = m(x) = \chi + \mu x, \qquad \text{where } \mu = \frac{b' - a'}{b - a}, \chi = b' - b\frac{b' - a'}{b - a}.$$

These are the parameters returned by the polynomial's `mapparms` function:

```
In [x]: chi, mu = p.mapparms()
In [x]: print(chi, mu)
-2001.0, 20.0
```

Therefore,

```
In [x]: print(q(chi + mu*100.05))
1.49422551
```

It is possible to change domain and window by direct assignment:

```
In [x]: q.domain = np.array((100., 100.1))
In [x]: print(q(100.05))
1.49422551
```

To evaluate a polynomial on a set number of evenly distributed points in its domain, for example, to plot it, use the Polynomial's linspace method:

```
In [x]: p.linspace(5)
Out [x]:
(array([ 100.   ,  100.025,  100.05 ,  100.075,  100.1  ]),
 array([ 1.80280222,  2.63107256,  1.49422551,  0.54527422,  0.39490249]))
```

p.linspace returns two arrays with the specified number of samples on the polynomial's domain representing the x points and the values the polynomial takes at those points, $p(x)$.

Polynomial.fit

The Polynomial method fit returns a least squares fitted polynomial to data, y, sampled at values x. In its simplest use, fit needs only to be passed array-like objects x and y, and a value for deg, the degree of polynomial to fit. It returns the polynomial which minimizes the sum of the squared errors,

$$E = \sum_i |y_i - p(x_i)|^2$$

For example,

```
In [x]: x = np.linspace(400, 700, 1000)
In [x]: y = 1 / x**4
In [x]: p = Polynomial.fit(x, y, 3)
```

produces the best-fit cubic polynomial to the function x^{-4} on the interval $(400, 700)$.

Weighted least-squares fitting is achieved by setting the argument, w, to a sequence of weighting values that is the same length as x and y. The polynomial returned is that which minimizes the sum of the *weighted* squared errors,

$$E = \sum_i w_i^2 |y_i - p(x_i)|^2$$

The domain and window of the fitted polynomial may be specified with the arguments domain and window; by default a minimal domain covering the points x is used.

It is wise to check the *quality* of the fit before using the returned polynomial. Setting the argument full=True causes fit to return two objects: the fitted polynomial and a list of various statistics about the fit itself:

```
In [x]: deg = 3
In [x]: p, [resid, rank, sing_val, rcond] = Polynomial.fit(x, y, deg, full=True)
In [x]: p
Out[x]:
Polynomial([  1.07041864e-11,  -1.16488662e-11,   1.02545751e-11,
        -5.64068914e-12], [ 400.,   700.], [-1.,   1.])
```

```
In [x]: resid
Out[x]: array([  4.57180972e-23])

In [x]: rank
Out[x]: 4

In [x]: sing_val
Out[x]: array([ 1.3843828 ,  1.32111941,  0.50462215,  0.28893641])

In [x]: rcond
Out[x]: 2.2204460492503131e-13
```

This list can be analyzed to see how well the polynomial function fits the data. `resid` is the sum of the squared residuals,

$$\mathtt{resid} = \sum_i |y_i - p(x_i)|^2$$

– a smaller value indicates a better fit. `rank` and `sing_val` are the rank and singular values of the matrix inverted in the least squares algorithm to find the polynomial coefficients: ill-conditioning of this matrix can lead to poor fits (particularly if the fitted polynomial degree is too high). `rcond` is the cutoff ratio for small singular values within this matrix: values smaller than this value are set to zero in the fit (to protect the fit from spurious artifacts introduced by round-off error) and a `RankWarning` exception is raised. If this happens, the data may be too noisy or not well described by the polynomial of the specified degree. Note that least squares fitting should always be carried out at double precision and be aware of "over-fitting" the data (attempting to fit a function with too many coefficients, i.e., a polynomial of too high order).

Example E6.11 A straight-line best fit is just a special case of a polynomial least squares fit (with `deg=1`). Consider the following data giving the absorbance over a path length of 5 mm of UV light at 280 nm, A, by a protein as a function of the concentration, [P]:

[P] / μg/mL	A
0	2.287
20	3.528
40	4.336
80	6.909
120	8.274
180	12.855
260	16.085
400	24.797
800	49.058
1500	89.400

We expect the absorbance to be linearly related to the protein concentration: $A = m[\mathrm{P}] + A_0$ where A_0 is the absorbance in the absence of protein (e.g., due to the solvent and experimental components).

Listing 6.8 Straight line fit to absorbance data

```
# eg6-polyfit.py
import numpy as np
import pylab
Polynomial = np.polynomial.Polynomial

# The data: conc = [P] and absorbance, A
conc = np.array([0, 20, 40, 80, 120, 180, 260, 400, 800, 1500])
A = np.array([2.287, 3.528, 4.336, 6.909, 8.274, 12.855, 16.085, 24.797,
              49.058, 89.400])

cmin, cmax = min(conc), max(conc)
pfit, stats = Polynomial.fit(conc, A, 1, full=True, window=(cmin, cmax),
                                                    domain=(cmin, cmax))

print('Raw fit results:', pfit, stats, sep='\n')

A0, m = pfit
resid, rank, sing_val, rcond = stats
rms = np.sqrt(resid[0]/len(A))

print('Fit: A = {:.3f}[P] + {:.3f}'.format(m, A0),
      '(rms residual = {:.4f})'.format(rms))

pylab.plot(conc, A, 'o', color='k')
pylab.plot(conc, pfit(conc), color='k')
pylab.xlabel('[P] /$\mathrm{\mu g\cdot mL^{-1}}$')
pylab.ylabel('Absorbance')
pylab.show()
```

The output shows a good straight-line fit to the data (Figure 6.8):

```
Raw fit results:
poly([ 1.92896129  0.0583057 ])
[array([ 2.47932733]), 2, array([ 1.26633786,  0.62959385]), 2.2204460492503131e-15]
Fit: A = 0.058[P] + 1.929 (rms residual = 0.4979)
```

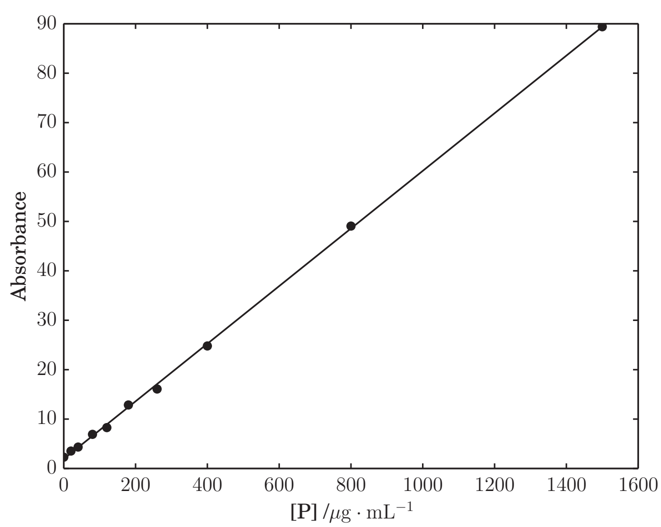


Figure 6.8 Line of least squares best fit to absorbance data as a function of concentration.

Table 6.6 Radius of the ball of fire produced by the "Trinity" nuclear test as a function of time

t /ms	R /m	t /ms	R /m	t /ms	R /m
0.1	11.1	1.36	42.8	4.34	65.6
0.24	19.9	1.50	44.4	4.61	67.3
0.38	25.4	1.65	46.0	15.0	106.5
0.52	28.8	1.79	46.9	25.0	130.0
0.66	31.9	1.93	48.7	34.0	145.0
0.80	34.2	3.26	59.0	53.0	175.0
0.94	36.3	3.53	61.1	62.0	185.0
1.08	38.9	3.80	62.9		
1.22	41.0	4.07	64.3		

Note: This data can be downloaded from scipython.com/ex/afg.

6.4.7 Exercises

Questions

Q6.4.1 The third derivative of the polynomial function $P(x) = 3x^3 + 2x - 7$ is 18, so why does the following evaluate as `False`?

```
In [x]: Polynomial((-7, 2, 0, 3)).deriv(3) == 18
Out[x]: False
```

Q6.4.2 Find and classify the stationary points of the polynomial

$$f(x) = (x^2 + x - 11)^2 + (x^2 + x - 7)^2.$$

Problems

P6.4.1 The expansion of the spherical ball of fire generated in an explosion may be analyzed to deduce the initial energy, E, released by a nuclear weapon. The British physicist Geoffrey Taylor used dimensional analysis to demonstrate that the radius of this sphere, $R(t)$, should be related to E, the air density, ρ_{air}, and time, t, through

$$R(t) = CE^{\frac{1}{5}}\rho_{\mathrm{air}}^{-\frac{1}{5}}t^{-\frac{2}{5}},$$

where, using model-shock wave problems, Taylor estimated the dimensionless constant $C \approx 1$. Using the data obtained from declassified timed images of the first New Mexico atomic explosion, Taylor confirmed this law and produced an estimate of the (then unknown) value of E. Use a log-log plot to fit the data in Table 6.6[24] to the model and confirm the time-dependence of R. Taking $\rho_{\mathrm{air}} = 1.25\ \mathrm{kg\,m^{-3}}$ deduce E and express its value in Joules and in "kilotons of TNT" where the explosive energy released by 1 ton of TNT is arbitrarily defined to be 4.184×10^9 J.

P6.4.2 Find the mean and variance of both x and y, the correlation coefficient and the equation of the linear regression line for each of the four data sets given in Table 6.7. Comment on these values in the light of a plot of the data.

[24] G. I. Taylor, (1950) *Proc. Roy. Soc. London* **A201**, 159.

Table 6.7 Four sample data sets for analysis of mean, variance and correlation

x_1	y_1	x_2	y_2	x_3	y_3	x_4	y_4
10.0	8.04	10.0	9.14	10.0	7.46	8.0	6.58
8.0	6.95	8.0	8.14	8.0	6.77	8.0	5.76
13.0	7.58	13.0	8.74	13.0	12.74	8.0	7.71
9.0	8.81	9.0	8.77	9.0	7.11	8.0	8.84
11.0	8.33	11.0	9.26	11.0	7.81	8.0	8.47
14.0	9.96	14.0	8.10	14.0	8.84	8.0	7.04
6.0	7.24	6.0	6.13	6.0	6.08	8.0	5.25
4.0	4.26	4.0	3.10	4.0	5.39	19.0	12.50
12.0	10.84	12.0	9.13	12.0	8.15	8.0	5.56
7.0	4.82	7.0	7.26	7.0	6.42	8.0	7.91
5.0	5.68	5.0	4.74	5.0	5.73	8.0	6.89

These data can be downloaded as the file `ex6-4-a-anscombe.tex` from scipython.com/ex/aff.

P6.4.3 The van der Waals equation of state may be written as follows to give the pressure, p, of a gas from its molar volume, V, and temperature, T:

$$p = \frac{RT}{V-b} - \frac{a}{V^2},$$

where a and b are molecule-specific constants and $R = 8.314\ \mathrm{J\,K^{-1}\,mol^{-1}}$ is the gas constant. It can readily be rearranged to yield the temperature for a given pressure and volume, but its form giving the molar volume in terms of pressure and temperature is a cubic equation:

$$pV^3 - (pb + RT)V^2 + aV - ab = 0$$

Of the three roots to this equation, below the *critical point*, (T_c, p_c) all are real: the largest and smallest give the molar volume of the gas phase and liquid phase respectively; above the critical point, where no liquid phase exists, only one root is real and gives the molar volume of the gas (also known in this region as a *supercritical fluid*). The critical point is given by the condition $(\partial p/\partial V)_T = (\partial^2 p/\partial V^2)_T = 0$ and for a van der Waals gas is given by the formulas

$$T_c = \frac{8a}{27Rb}, \qquad p_c = \frac{a}{27b^2}$$

For ammonia the van der Waals constants are $a = 4.225\ \mathrm{L^2\,bar\,mol^{-2}}$ and $b = 0.03707\ \mathrm{L\,mol^{-1}}$.

a. Find the critical point of ammonia, and then determine the molar volume at room temperature and pressure, (298 K, 1 atm) and at (500 K, 12 MPa).
b. An *isotherm* is the set of points (p, V) at a constant temperature satisfying an equation of state. Plot the isotherm (p against V) for ammonia at 350 K using the

van der Waals equation of state and compare it with the 350 K isotherm for an ideal gas, which has the equation of state $p = RT/V$.

P6.4.4 The first-stage rockets of the Saturn V rocket that launched the Apollo 11 mission generated an acceleration which increases with time throughout their operation (mostly because of the decrease in mass as it burns its fuel). This acceleration may be modeled (in units of $\mathrm{m\,s^{-2}}$) as a function of time after launch, t in seconds, by the quadratic function:

$$a(t) = 2.198 + (2.842 \times 10^{-2})t + (1.061 \times 10^{-3})t^2$$

Determine the distance traveled by the rocket at the end of the stage-one center-engine burn, 2 minutes, 15.2 seconds, after launch.

(Harder) Assuming a constant lapse rate of $\Gamma = -\mathrm{d}T/\mathrm{d}z = 6\ \mathrm{K\,km^{-1}}$ and a ground temperature of 302 K, at what time and altitude, z, did the rocket achieve Mach 1? During the relevant phase of the launch, take the average pitch angle to be 12°, and assume the speed of sound can be calculated as a function of absolute temperature to be

$$c = \sqrt{\frac{\gamma RT}{M}},$$

where the constant $\gamma = 1.4$ and the mean molar mass of the atmosphere is $M = 0.0288\ \mathrm{kg\,mol^{-1}}$.

6.5 Linear algebra

6.5.1 Basic matrix operations

Although NumPy does have a `matrix` object (see Section 6.6), all the same matrix operations can be carried out on a regular two-dimensional NumPy array. These include scalar multiplication, matrix (dot) product, elementwise multiplication and transpose:

```
In [x]: A = np.array([[0, 0.5], [-1, 2]])
In [x]: A
Out[x]:
array([[ 0. ,  0.5],
       [-1. ,  2. ]])
In [x]: A * 5                # multiplication by a scalar
Out[x]:
array([[  0. ,   2.5],
       [ -5. ,  10. ]])
In [x]: B = np.array([[2, -0.5], [3, 1.5]])

In [x]: B
Out[x]:
array([[ 2. , -0.5],
       [ 3. ,  1.5]])

In [x]: A.dot(B)             # or np.dot(A,B): matrix product
```

```
Out[x]:
array([[ 1.5 ,  0.75],
       [ 4.  ,  3.5 ]])

In [x]: A * B              # elementwise multiplication
Out[x]:
array([[ 0.  , -0.25],
       [-3.  ,  3.  ]])
In [x]: A.transpose()      # or simply A.T
Out[x]:
array([[ 0. , -1. ],
       [ 0.5,  2. ]])
```

Note that the transpose returns a *view* on the original matrix.

The identity matrix is returned by passing the two dimensions of the matrix to the method `np.eye`:

```
In [x]: np.eye(3,3)
Out[x]:
array([[ 1.,  0.,  0.],
       [ 0.,  1.,  0.],
       [ 0.,  0.,  1.]])
```

Matrix products

NumPy contains further methods for vector and matrix products. For example,

```
In [x]: a = np.array([1,2,3])
In [x]: b = np.array([0,1,2])
In [x]: np.inner(a,b)         # inner product; here, the same as a.dot(b)
Out[x]: 8

In [x]: np.outer(a,b)         # outer product
Out[x]:
array([[0, 1, 2],
       [0, 2, 4],
       [0, 3, 6]])
```

To raise a matrix to an (integer) power, however, requires a method from the `np.linalg` module:

```
In [x]: A = np.array([[0, 0.5], [-1, 2]])
In [x]: np.linalg.matrix_power(A, 3)        # the same as A.dot(A.dot(A))
Out[x]:
array([[-1.  ,  1.75],
       [-3.5 ,  6.  ]])
```

Note that the `**` operator performs *elementwise* exponentiation:

```
In [x]: A**3                  # the same as A * A * A
Out[x]:
array([[ 0.   ,  0.125],
       [-1.   ,  8.   ]])
```

Other matrix properties

The norm of a matrix or vector is returned by the function `np.linalg.norm`. It is possible to calculate several different norms (see the documentation), but the ones used

by default, are the Frobenius norm for two-dimensional arrays:

$$||A|| = \left(\sum_{i,j} |a_{ij}|^2 \right)^{1/2}$$

and the Euclidean norm for one-dimensional arrays:

$$||a|| = \left(\sum_{i} |z_i|^2 \right)^{1/2} = \sqrt{|z_0|^2 + |z_1|^2 + \cdots + |z_{n-1}|^2}.$$

Thus,

```
In [x]: np.linalg.norm(A)
Out[x]: 2.2912878474779199

In [x]: c = np.array([1, 2j, 1 - 1j])
In [x]: np.linalg.norm(c)
Out[x]: 2.6457513110645907      # sqrt(1 + 4 + 2)
```

The function `np.linalg.det` returns the determinant of a matrix, and the regular NumPy function `np.trace` returns its trace (the sum of its diagonal elements):

```
In [x]: np.linalg.det(A)
Out[x]: 0.5

In [x]: np.trace(A)
Out[x]: 2.0
```

The *rank* of a matrix is obtained using `np.linalg.matrix_rank`:

```
In [x]: np.linalg.matrix_rank(A)        # matrix A has full rank
Out[x]: 2
In [x]: D = np.array([[1,1],[2,2]])     # a rank deficient matrix

In [x]: np.linalg.matrix_rank(D)
Out[x]: 1
```

To find the inverse of a square matrix, use `np.linalg.inv`. A `LinAlgError` exception is raised if the matrix inversion fails:

```
In [x]: np.linalg.inv(A)
Out[x]:
array([[ 4., -1.],
       [ 2.,  0.]])

In [x]: np.linalg.inv(D)
...
LinAlgError: Singular matrix
```

6.5.2 Eigenvalues and eigenvectors

To calculate the eigenvalues and (right) eigenvectors of a general square array with shape `(n,n)`, use `np.linalg.eig`, which returns the eigenvalues, `w`, as an array of shape `(n,)` and the normalized eigenvectors, `v`, as a complex array of shape `(n,n)`.

The eigenvalues are not returned in any particular order, but the eigenvalue `w[i]` corresponds to the eigenvector `v[:,i]`. Note that the eigenvectors are arranged in columns. If the eigenvalue calculation does not converge for some reason a `LinAlgError` is raised.

```
In [x]: vals, vecs = np.linalg.eig(A)
In [x]: vals
Out[x]: array([ 0.29289322,  1.70710678])

❶ In [x]: np.isclose(np.sum(vals), A.trace())
Out[x]: True

In [x]: vecs
Out[x]:
array([[-0.86285621, -0.28108464],
       [-0.50544947, -0.95968298]])
```

❶ Verify that the sum of the eigenvalues is equal to the matrix trace.

If the matrix is Hermitian or real-symmetric, the function `np.linalg.eigh` may be used instead. This method takes an additional argument, `UPLO`, which can be `'L'` or `'U'` according to whether the lower or upper triangular part of the matrix is used. The default is `'L'`.

Two additional methods, `np.linalg.eigvals` and `np.linalg.eigvalsh`, return *only the eigenvalues* (and not the eigenvectors) of a general and Hermitian matrix respectively.

Since NumPy version 1.8, these and most other `linalg` methods follow the usual broadcasting rules so that several matrices can be operated on at once: each matrix is assumed to be stored in the last two dimensions. For example, we may work with an array with shape `(3,2,2)` representing the three 2×2 Pauli matrices:

$$\sigma_x = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \quad \sigma_y = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix} \quad \sigma_z = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$$

```
In [x]: pauli_matrices = np.array((
                ((0, 1), (1, 0)),       # sigma_x
                ((0, -1j), (1j, 0)),    # sigma_y
                ((1, 0), (0, -1))       # sigma_z
                ))
In [x]: np.linalg.eigh(pauli_matrices)
Out[x]:
(array([[-1.,  1.],
        [-1.,  1.],
        [-1.,  1.]]),
 array([[[-0.70710678+0.j        ,  0.70710678+0.j        ],
         [ 0.70710678+0.j        ,  0.70710678+0.j        ]],

        [[-0.70710678-0.j        , -0.70710678+0.j        ],
         [ 0.00000000+0.70710678j,  0.00000000-0.70710678j]],

        [[ 0.00000000+0.j        ,  1.00000000+0.j        ],
         [ 1.00000000+0.j        ,  0.00000000+0.j        ]]]))
```

6.5.3 Solving equations

Linear scalar equations

NumPy provides an efficient and numerically stable method for solving systems of linear scalar equations. The set of equations

$$
\begin{aligned}
m_{11}x_1 + m_{12}x_2 + \cdots + m_{1n}x_1 &= b_1 \\
m_{21}x_1 + m_{22}x_2 + \cdots + m_{2n}x_2 &= b_2 \\
&\cdots \\
m_{n1}x_1 + m_{n2}x_2 + \cdots + m_{nn}x_n &= b_n
\end{aligned}
$$

can be expressed as the matrix equation $\mathbf{Mx} = \mathbf{b}$:

$$
\begin{pmatrix} m_{11} & m_{12} & \cdots & m_{1n} \\ m_{21} & m_{22} & \cdots & m_{2n} \\ \vdots & & \ddots & \vdots \\ m_{n1} & m_{n2} & \cdots & m_{nn} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix}.
$$

The solution of this system of equations (the vector $\mathbf{x}$) is returned by the `np.linalg.solve` method. For example, the three simultaneous equations

$$
\begin{aligned}
3x - 2y &= 8 \\
-2x + y - 3z &= -20 \\
4x + 6y + z &= 7
\end{aligned}
$$

can be represented as the matrix equation $\mathbf{Mx} = \mathbf{b}$:

$$
\begin{pmatrix} 3 & -2 & 0 \\ -2 & 1 & -3 \\ 4 & 6 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} 8 \\ -20 \\ 7 \end{pmatrix}
$$

and solved by passing arrays corresponding to matrix $\mathbf{M}$ and vector $\mathbf{b}$ to `np.linalg.solve`:

```
In [x]: M = np.array([[3,-2,0],[-2,1,-3],[4,6,1]])
In [x]: b = np.array([8,-20,7])
In [x]: np.linalg.solve(M, b)
Out[x]: array([ 2., -1.,  5.])
```

That is, $x = 2, y = -1, z = 5$.

If no unique solution exists (for nonsquare or singular matrix, $\mathbf{M}$), a `LinAlgError` is raised.

Linear least squares solutions ("best fit")

Where a set of equations, $\mathbf{Mx} = \mathbf{b}$, does not have a unique solution, a least squares solution that minimizes the L^2 norm, $||b - Mx||^2$ (sum of squared residuals) may be sought using the `np.linalg.lstsq` method. This is the type of problem described as *over-determined* (more data points than the two unknown quantities, m and c). Passed

`M` and `b`, `np.linalg.lstsq` returns the solution array `x`, the sum of squared residuals, the rank of `M` and the singular values of `M`.

A typical use of this method is to find the "line of best-fit", $y = mx + c$, through some data thought to be linearly related as in the following example.

Example E6.12 The Beer-Lambert Law relates the concentration, c, of a substance in a solution sample to the intensity of light transmitted through the sample, I_t across a given path length, l, at a given wavelength, λ:

$$I_\mathrm{t} = I_0 e^{-\alpha c l},$$

where I_0 is the incident light intensity and α is the absorption coefficient at λ.

Given a series of measurements of the fraction of light transmitted, I_t/I_0, α may be determined through a least squares fit to the straight line:

$$y = \ln \frac{I_\mathrm{t}}{I_0} = -\alpha c l.$$

Although this line passes through the origin ($y = 0$ for $c = 0$), we will fit the more general linear relationship:

$$y = mc + k$$

where $m = -\alpha l$, and verify that k is close to zero.

Given a sample with path length $l = 0.8$ cm, the following data were measured for I_t/I_0 at five different concentrations:

c /M	I_t/I_0
0.4	0.886
0.6	0.833
0.8	0.784
1.0	0.738
1.2	0.694

The matrix form of the least squares equation to be solved is

$$\begin{pmatrix} c_1 & 1 \\ c_2 & 1 \\ c_3 & 1 \\ c_4 & 1 \\ c_5 & 1 \end{pmatrix} \begin{pmatrix} m \\ k \end{pmatrix} = \begin{pmatrix} T_1 \\ T_2 \\ T_3 \\ T_4 \\ T_5 \end{pmatrix}$$

where $T = \ln(I_\mathrm{t}/I_0)$. The code here determines m and hence α using `np.linalg.lstsq`:

Listing 6.9 Linear least squares fitting of the Beer-Lambert Law

```
# eg6-beer-lambert-lstsq.py
import numpy as np
import pylab
```

```
# Path length, cm
path = 0.8
# The data: concentrations (M) and It/I0
c = np.array([0.4, 0.6, 0.8, 1.0, 1.2])
It_over_I0  = np.array([ 0.891 ,  0.841,  0.783,  0.744,  0.692])

n = len(c)
A = np.vstack((c, np.ones(n))).T
T = np.log(It_over_I0)

x, resid, _, _ = np.linalg.lstsq(A, T)    ❶
m, k = x
alpha = - m / path
print('alpha = {:.3f} M-1.cm-1'.format(alpha))
print('k =', k)
print('rms residual = ', np.sqrt(resid[0]))

pylab.plot(c, T, 'o')
pylab.plot(c, m*c + k)
pylab.xlabel('$c\;/\mathrm{M}$')
pylab.ylabel('$\ln(I_\mathrm{t}/I_0)$')
pylab.show()
```

❶ Here, _ is the dummy variable name conventionally given to an object we do not need to store or use.

The output produces a best fit value of $\alpha = 0.393\ \mathrm{M^{-1}\,cm^{-1}}$ and a value of k compatible with experimental error:

```
alpha = 0.393 M-1.cm-1
k = 0.0118109033334
rms residual =  0.0096843591966
```

Figure 6.9 shows the data and fitted line.

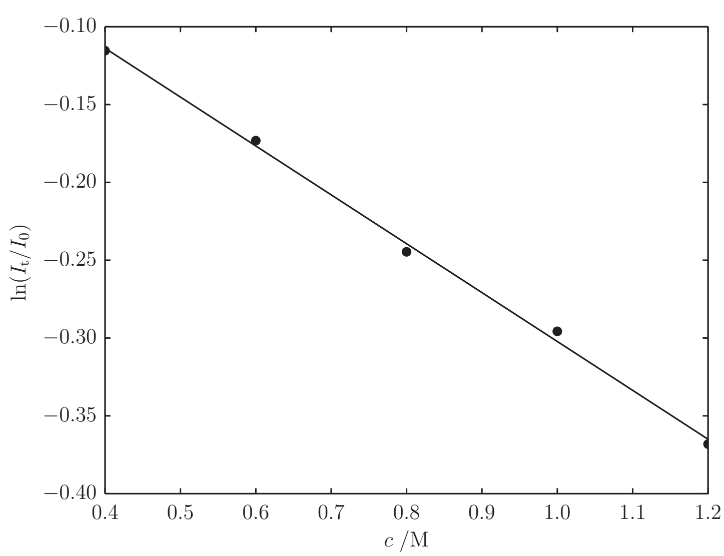


Figure 6.9 Line of least squares best fit to absorbance data as a function of concentration.

6.5.4 Exercises

Questions

Q6.5.1 Demonstrate that the three Pauli matrices given in Section 6.5.2 are *unitary*. That is, that $\sigma_p^\dagger \sigma_p = I_2$ for $p = x, y, z$, where I_2 is the 2×2 identity matrix and $\dagger$ denotes the Hermitian conjugate (conjugate transpose).

Q6.5.2 The ticker timer, much used in school physics experiments, is a device that marks dots on a strip of paper tape at evenly spaced intervals of time as the tape moves through it at some (possibly variable) speed. The following data relate to the positions (in cm) of marks on a tape pulled through a ticker timer by a falling weight. The marks are made every 1/10 sec.

```
x = [1.3, 6.0, 20.2, 43.9, 77.0, 119.6, 171.7, 233.2, 304.2, 384.7,
     474.7, 574.1, 683.0, 801.3, 929.2, 1066.4, 1213.2, 1369.4, 1535.1,
     1710.3, 1894.9]
```

Fit these data to the function $x = x_0 + v_0 t + \frac{1}{2}gt^2$ and determine an approximate value for the acceleration due to gravity, g.

Problems

P6.5.1 In physics, the *Planck units* of measurement are those defined such that the five universal physical constants, c (the speed of light), G (the gravitational constant), $\hbar$ (the reduced Planck constant), $(4\pi\epsilon_0)^{-1}$ (the Coulomb constant) and k_B (the Boltzmann constant) are set to unity. The dimensions of these quantities in terms of length (L), mass (M), time (T), charge (Q) and thermodynamic temperature (Θ) are given in Table 6.8, along with their values in SI units.

This suggests the following matrix relationship between the constants and their dimensions:

$$
\begin{array}{c}
c \\ G \\ \hbar \\ (4\pi\epsilon_0)^{-1} \\ k_\mathrm{B}
\end{array}
\begin{array}{c}
\begin{array}{ccccc} \mathrm{L} & \mathrm{M} & \mathrm{T} & \mathrm{Q} & \Theta \end{array} \\
\begin{pmatrix}
1 & 0 & -1 & 0 & 0 \\
3 & -1 & -2 & 0 & 0 \\
2 & 1 & -1 & 0 & 0 \\
3 & 1 & -2 & -2 & 0 \\
2 & 1 & -2 & 0 & -1
\end{pmatrix}
\end{array}
$$

Table 6.8 Some physical constants and their dimensions

c	Speed of light	$2.99792458 \times 10^{8}\ \mathrm{m\,s^{-1}}$	$\mathrm{L\,T^{-1}}$
G	Gravitational constant	$6.67384 \times 10^{-11}\ \mathrm{m^3\,kg^{-1}\,s^{-2}}$	$\mathrm{L^3\,M^{-1}\,T^{-2}}$
$\hbar$	Reduced Planck constant	$1.054571726 \times 10^{-34}\ \mathrm{J\,s}$	$\mathrm{L^2\,M\,T^{-1}}$
$(4\pi\epsilon_0)^{-1}$	Coulomb constant	$8.9875517873681764 \times 10^{9}\ \mathrm{N\,m^2\,C^{-2}}$	$\mathrm{L^3\,M\,T^{-2}\,Q^{-2}}$
k_B	Boltzmann constant	$1.3806488 \times 10^{-23}\ \mathrm{J\,K^{-1}}$	$\mathrm{L^2\,M\,T^{-2}\,\Theta^{-1}}$

Using the inverse of this matrix, determine the SI values of length, mass, time, charge and temperature in the base Planck units; that is, the combination of these physical constants yielding the dimensions L, M, T, Q and Θ. For example, the *Planck length* is found to be $l_\mathrm{P} = \sqrt{\hbar G/c^3} = 1.616199 \times 10^{-35}$ m.

P6.5.2 The (symmetric) matrix representing the inertia tensor of a collection of masses, m_i, with positions (x_i, y_i, z_i) relative to their center of mass is

$$\mathbf{I} = \begin{pmatrix} I_{xx} & I_{xy} & I_{xz} \\ I_{xy} & I_{yy} & I_{yz} \\ I_{xz} & I_{yz} & I_{zz} \end{pmatrix},$$

where

$$I_{xx} = \sum_i m_i(y_i^2 + z_i^2), \qquad I_{yy} = \sum_i m_i(x_i^2 + z_i^2), \qquad I_{zz} = \sum_i m_i(x_i^2 + y_i^2),$$
$$I_{xy} = -\sum_i m_i x_i y_i, \qquad I_{yz} = -\sum_i m_i y_i z_i, \qquad I_{xz} = -\sum_i m_i x_i z_i.$$

There exists a transformation of the coordinate frame such that this matrix is diagonal: the axes of this transformed frame are called the *principal* axes and the diagonal inertia matrix elements, $I_a \le I_b \le I_c$, are the *principal moments of inertia*.

Write a program to calculate the principal moments of inertia of a molecule, given the position and masses of its atoms *relative to some arbitrary origin*. Your program should first relocate the atom coordinates relative to its center of mass and then determine the principal moments of inertia as the eigenvalues of the matrix **I**.

A molecule may be classified as follows according to the relative values of I_a, I_b and I_c:

- $I_a = I_b = I_c$: spherical top;
- $I_a = I_b < I_c$: oblate symmetric top;
- $I_a < I_b = I_c$: prolate symmetric top;
- $I_a < I_b < I_c$: asymmetric top.

Determine the principal moments of inertia and classify the molecules NH_3, CH_4, CH_3Cl and O_3 given the data available at scipython.com/ex/afh. Also determine the *rotational constants*, A, B and C, related to the moments of inertia through $Q = h/(8\pi^2 c I_q)$ $(Q = A, B, C; q = a, b, c)$ and usually expressed in cm^{-1}.

◊ **P6.5.3** The NumPy method `numpy.linalg.svd` returns the *singular value decomposition* (SVD) of a matrix, **M**, as the arrays **U**, Σ and **V** satisfying the factorization: $\mathbf{M} = \mathbf{U}\Sigma\mathbf{V}^\dagger$ where † denotes the Hermitian conjugate (the conjugate transpose).

The SVD and the eigendecomposition are related in that the left-singular row vectors, **U** are the eigenvectors of $\mathbf{MM}^*$ and the right-singular column vectors, **V**, are the eigenvectors of $\mathbf{M}^*\mathbf{M}$. Furthermore, the diagonal entries of Σ are the square roots of the nonzero eigenvalues of both $\mathbf{MM}^*$ and $\mathbf{MM}^*$.

Show that this is the case for the special case of **M** a 3 × 3 matrix with random real entries by comparing the output of `numpy.linalg.svd` with that of `numpy.linalg.eig`.

Hint: the singular values of **M** are sorted in descending order, but the eigenvalues returned by `numpy.linalg.eig` are in no particular order. Both methods produce normalized eigenvectors, but may differ by sign (ignore the possibility that any of the eigenvalues could have an eigenspace with dimension greater than 1).

6.6 Matrices

The NumPy `matrix` class is a subclass of the regular `ndarray` which provides some convenient functionality for dealing with matrices. There are some important differences from conventional arrays, and care should be taken in using some of the familiar array operations as they have been overridden in the matrix subclass and behave differently.

A matrix is *always a two-dimensional array*. Even a row or column matrix has two dimensions (with shape `(1,n)` or `(n,1)` respectively), and flattening a matrix with `flatten` or `ravel` (see Section 6.1.5) returns a `(1,n)` array rather than a one-dimensional array.

6.6.1 Creating a matrix

As an alternative to the regular `ndarray` construction methods, a `matrix` object can be created using the MATLAB-like syntax using a string of values in which columns are separated by spaces and rows by semicolons:

```
In [x]: A = np.matrix([[0, -1], [1, -2]])     # as for np.array()
In [x]: B = np.matrix('0 -1; 1 -2')           # MATLAB-like
In [x]: print(B)
[[ 0 -1]
 [ 1 -2]]
```

The data type of the matrix can be set with the `dtype` attribute as for regular arrays. If a matrix is created from an existing `ndarray` object, the default behavior is to *copy* the data into the new matrix object; to construct a matrix consisting of a *view* on an existing `ndarray`'s data, set the attribute `copy=False`:

```
In [x]: a = np.array([[1, 2], [3, 4]])
In [x]: A = np.matrix(a, copy=False)
In [x]: B = np.matrix(a)
In [x]: a[0,0] = -1
In [x]: print(A[0,0], B[0,0])
-1 1
```

That is, `A` is updated by the assignment `a[0,0]= -1`, but `B` owns its own data and is not updated. Special matrices such as the identity matrix are best created using the corresponding `ndarray` constructor and passing the resulting array object to `matrix` with `copy=False`:

```
In [x]: I = np.matrix(np.eye(2,2), copy=False)
In [x]: N = np.matrix(np.zeros((2,2)), copy=False)
In [x]: W = np.matrix(np.ones((2,2)), copy=False)
```

Example E6.13 One way to create the two-dimensional rotation matrix,

$$\mathbf{R} = \begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix}$$

which rotates points in the xy plane counterclockwise through $\theta = 30°$ about the origin:

```
In [x]: theta = np.radians(30)
In [x]: c, s = np.cos(theta), np.sin(theta)
In [x]: R = np.matrix('{} {}; {} {}'.format(c, -s, s, c))
In [x]: print(R)
[[ 0.8660254 -0.5      ]
 [ 0.5        0.8660254]]
```

6.6.2 Matrix operations

The most important difference between `matrix` objects and arrays is in the behavior of the `*` and `**` operators. As we have seen, these act *elementwise* on `ndarrays`:

```
In [x]: a = np.array([[0, -1], [1, -2]])
In [x]: a * a       # arrays: elementwise (Hadamard) product
Out[x]:
array([[0, 1],
       [1, 4]])
In [x]: a ** 3      # arrays: elementwise exponentiation
Out[x]:
array([[ 0, -1],
       [ 1, -8]])
```

That is,

$$\begin{pmatrix} 0 & -1 \\ 1 & -2 \end{pmatrix} \circ \begin{pmatrix} 0 & -1 \\ 1 & -2 \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 4 \end{pmatrix},$$

$$\begin{pmatrix} 0^3 & -1^3 \\ 1^3 & -2^3 \end{pmatrix} = \begin{pmatrix} 0 & -1 \\ 1 & -8 \end{pmatrix}.$$

For `matrix` objects these operators are *matrix* multiplication and exponentiation:

```
In [x]: A = np.matrix([[0, -1], [1, -2]])
In [x]: A * A       # matrix multiplication
matrix([[-1,  2],
        [-2,  3]])
In [x]: A ** 3      # ie A.A.A
matrix([[ 2, -3],
        [ 3, -4]])
```

That is,

$$\begin{pmatrix} 0 & -1 \\ 1 & -2 \end{pmatrix}\begin{pmatrix} 0 & -1 \\ 1 & -2 \end{pmatrix} = \begin{pmatrix} -1 & 2 \\ -2 & 3 \end{pmatrix},$$

and

$$\begin{pmatrix} 0 & -1 \\ 1 & -2 \end{pmatrix}^3 = \begin{pmatrix} 0 & -1 \\ 1 & -2 \end{pmatrix}\begin{pmatrix} 0 & -1 \\ 1 & -2 \end{pmatrix}\begin{pmatrix} 0 & -1 \\ 1 & -2 \end{pmatrix} = \begin{pmatrix} 2 & -3 \\ 3 & -4 \end{pmatrix},$$

This simplifies the otherwise slightly cumbersome equivalents: `a.dot(a)` and `a.dot(a).dot(a)`. Note that for both `ndarray` and `matrix` objects, multiplication by a *scalar* acts elementwise:

```
In [x]: A * 4
matrix([[ 0, -4],
        [ 4, -8]])
```

As might be expected, matrix operations that already have methods implemented by the `ndarray` class are retained, including `transpose` (also available as the `T` attribute) and `diagonal`. Additionally, there are attributes for the Hermitian transpose (`H`) and matrix inverse (`I`). If the matrix is singular, a `LinAlgError` exception is raised if an attempt is made to take its inverse.

For eigenfunctions and eigenvalues, see the description of NumPy's `linalg` module (Section 6.5.2).

Example E6.14 The matrix **B**, defined here, may be manipulated as follows:

$$\mathbf{B} = \begin{pmatrix} 1 & 3-j \\ 3j & -1+j \end{pmatrix}, \quad \mathbf{B}^{\mathrm{T}} = \begin{pmatrix} 1 & 3j \\ 3-j & -1+j \end{pmatrix}$$

$$\mathbf{B}^{\dagger} = \begin{pmatrix} 1 & -3j \\ 3+j & -1-j \end{pmatrix}, \quad \mathbf{B}^{-1} = \begin{pmatrix} -\frac{1}{20} - \frac{3}{20}j & \frac{1}{20} - \frac{7}{20}j \\ \frac{3}{10} + \frac{3}{20}j & -\frac{1}{20} + \frac{1}{10}j \end{pmatrix}.$$

```
In [x]: B = np.matrix([[1, 3-1j], [3j, -1+1j]])
In [x]: print(B)
[[ 1.+0.j  3.-1.j]
 [ 0.+3.j -1.+1.j]]

In [x]: print(B.T)
[[ 1.+0.j  0.+3.j]
 [ 3.-1.j -1.+1.j]]

In [x]: print(B.H)
[[ 1.-0.j  0.-3.j]
 [ 3.+1.j -1.-1.j]]

In [x]: print(B.I)
[[-0.05-0.15j  0.05-0.35j]
 [ 0.30+0.15j -0.05+0.1j ]]
```

Note that although these derived matrices look like attributes, they are not calculated until requested,[25] and so the use of the `matrix` class is not significantly slower than using regular `ndarray`s.

A few other common matrix operations are found elsewhere in the NumPy package, including the trace, determinant, eigenvalues and (right) eigenvectors:

```
In [x]: print(np.trace(B))
1j
In [x]: print(np.linalg.det(B))
(-4-8j)
In [x]: eigenvalues, eigenvectors = np.linalg.eig(B)
In [x]: print(eigenvalues, eigenvectors, sep='\n\n')
[ 2.50851535+2.09456868j -2.50851535-1.09456868j]

[[ 0.77468569+0.j         -0.52924821+0.38116633j]
 [ 0.18832434+0.60365224j  0.75802940+0.j        ]]
```

6.6.3 Should you use NumPy matrices?

The NumPy Matrix class is convenient if you have a lot of operations to perform with matrices and like the MATLAB-style syntax for manipulating them, but it does not provide any functionality that isn't already available to `ndarray` objects. The multiplication operator, `*`, acting to produce matrix products can make code clearer but other common matrix operations still require the use of the main NumPy library's modules and functions. Indeed, the `matrix` class's insistence in turning everything into a two-dimensional array can be rather trying. For example, a $1 \times n$ row matrix must be indexed `M[0,j]` where $j = 0, 1, ..., n - 1$, and, bizarrely, even the `trace` method called on a matrix object returns a two-dimensional matrix object:

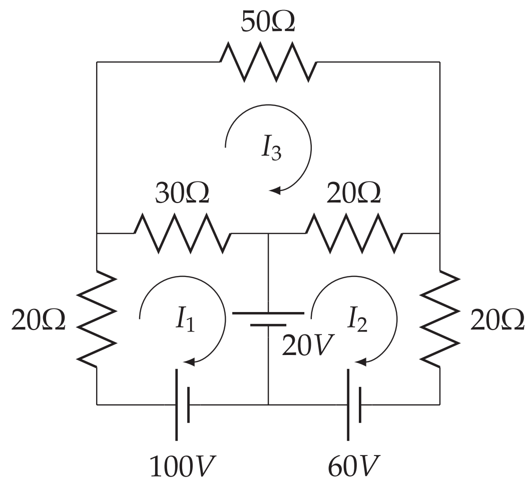
```
In [x]: A.trace()
matrix([[-2]])          # ?!
In [x]: np.trace(A)     # recommended alternative
-2
```

In short, while `matrix` objects may have the edge for simple calculations in an interactive session, they do not have much to commend them over regular `ndarray`s to any but the most die-hard MATLAB fans.[26]

Example E6.15 The currents flowing in the closed regions labeled I_1, I_2 and I_3 of the circuit given here may be analyzed by *mesh analysis*.

[25] They are *properties* of the `matrix` class which, in this case, are really class methods masquerading as attributes.

[26] Moreover, at the time of writing it seems that Python 3.5 is likely to include a specific infix operator for matrix multiplication, `@`.



For each closed loop, we can apply Kirchoff's Voltage Law ($\sum_k V_k = 0$) in conjunction with Ohm's Law ($V = IR$), to give three simultaneous equations:

$$50I_1 - 30I_3 = 80,$$
$$40I_2 - 20I_3 = 80,$$
$$-30I_1 - 20I_2 + 100I_3 = 0.$$

These can be expressed in matrix form as $\mathbf{RI} = \mathbf{V}$:

$$\begin{pmatrix} 50 & 0 & -30 \\ 0 & 40 & -20 \\ -30 & -20 & 100 \end{pmatrix} \begin{pmatrix} I_1 \\ I_2 \\ I_3 \end{pmatrix} = \begin{pmatrix} 80 \\ 80 \\ 0 \end{pmatrix},$$

We could use the numerically stable `np.linalg.solve` method (Section 6.5.3) to find the loop currents, **I** here, but in this well-behaved system[27], let's find them by left multiplication by the matrix inverse, $\mathbf{R^{-1}}$:

$$\mathbf{R^{-1}RI} = \mathbf{I} = \mathbf{R^{-1}V}.$$

Using NumPy's `matrix` module:

```
In [x]: R = np.matrix('50 0 -30; 0 40 -20; -30 -20 100')
In [x]: V = np.matrix('80; 80; 0')
In [x]: I = np.linalg.inv(R) * V
In [x]: print(I)
[[ 2.33333333]
 [ 2.61111111]
 [ 1.22222222]]
```

Thus, $I_1 = 2.33$ A, $I_2 = 2.61$ A, $I_3 = 1.22$ A.

[27] In general, matrix inversion may be an ill-conditioned problem, but this particular matrix is easy to invert accurately. See Section 9.2.2 for more on conditioning.

6.6.4 Exercises

Problems

P6.6.1 Let the column matrix

$$\mathbf{F_n} = \begin{pmatrix} p_n \\ q_n \end{pmatrix}$$

describe the number of non-negative integers less than 10^n ($n \geq 0$) that do (p_n) and do not (q_n) contain the digit 5. Hence, for $n = 1, p_1 = 1$ and $q_1 = 9$. Devise a matrix-based recursion relation for finding $\mathbf{F_{n+1}}$ from $\mathbf{F_n}$.

How many numbers less than 10^{10} contain the digit 5?

For each $n \leq 10$, find p_n and verify that $p_n = 10^n - 9^n$.

P6.6.2 The matrix

$$\mathbf{F} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$$

can be used to produce the Fibonacci sequence by repeated multiplication: the element F_{11}^n of the matrix $\mathbf{F}^n$ is the $(n+1)$th Fibonacci number (for $n = 0, 1, 2 \cdots$). Use NumPy's `matrix` objects to calculate the first 10 Fibonacci numbers.

One can show that

$$\mathbf{F}^n = \mathbf{C}\mathbf{D}^n\mathbf{C}^{-1}, \quad \text{where} \quad \mathbf{D} = \mathbf{C}^{-1}\mathbf{F}\mathbf{C}$$

is the diagonal matrix related to $\mathbf{F}$ through the similarity transformation associated with matrix `C`. Use this relationship to find the 1100th Fibonacci number.

P6.6.3 The *implicit* formula for a conic section may be written as the second-degree polynomial,

$$Q = Ax^2 + Bxy + Cy^2 + Dx + Ey + F = 0,$$

or in matrix form using the homogeneous coordinate vector,

$$\mathbf{x} = \begin{pmatrix} x \\ y \\ 1 \end{pmatrix},$$

as $\mathbf{x}^T\mathbf{Q}\mathbf{x} = 0$, where

$$\mathbf{Q} = \begin{pmatrix} A & B/2 & D/2 \\ B/2 & C & E/2 \\ D/2 & E/2 & F \end{pmatrix}.$$

Conic sections may be classified according to the following properties of $\mathbf{Q}$, where the submatrix $\mathbf{Q}_{33}$ is

$$\mathbf{Q}_{33} = \begin{pmatrix} A & B/2 \\ B/2 & C \end{pmatrix}.$$

- If $\det\mathbf{Q} = 0$, the conic is *degenerate* in one of the following forms:
  - if $\det\mathbf{Q}_{33} < 0$, the equation represents two intersecting lines,
  - if $\det\mathbf{Q}_{33} = 0$, the equation represents two parallel lines,
  - if $\det\mathbf{Q}_{33} > 0$, the equation represents a single point.
- If $\det\mathbf{Q} < 0$, the conic is a *hyperbola*.
- If $\det\mathbf{Q} > 0$, the conic is an *ellipse*:
  - If $A = C$ and $B = 0$, the ellipse is a *circle*.

Write a program to classify the conic section represented by the six coefficients A, B, C, D, E and F.

Some test-cases (coefficients not given are zero):

- Hyperbola: $B = 1, F = -9$.
- Parabola: $A = \frac{1}{2}, D = 2, E = -\frac{1}{2}$.
- Circle: $A = \frac{1}{2}, C = \frac{1}{2}, D = -2, E = -3, F = 2$.
- Ellipse: $A = 9, C = 4, F = -36$.
- Two parallel lines: $A = 1, F = -1$.
- A single point: $A = 1, C = 1$.

6.7 Random sampling

NumPy's `random` module provides methods for obtaining random numbers from any of several distributions as well as convenient ways to choose random entries from an array and to randomly shuffle the contents of an array.

As with the core library's `random` module (Section 4.5.1), `np.random` uses a Mersenne Twister *pseudorandom* number generator (PRNG). The way it seeds itself is operating-system dependent, but it can be reseeded with any hashable object (e.g., an immutable object such as an integer) by calling `np.random.seed`. For example, using the `randint` method described here:

```
In [x]: np.random.seed(42)
In [x]: np.random.randint(1, 10, 10)    # 10 random integers in [1,10)
array([7, 4, 8, 5, 7, 3, 7, 8, 5, 4])
In [x]: np.random.randint(1, 10, 10)
array([8, 8, 3, 6, 5, 2, 8, 6, 2, 5])
In [x]: np.random.randint(1, 10, 10)
array([1, 6, 9, 1, 3, 7, 4, 9, 3, 5])
In [x]: np.random.seed(42)              # reseed the PRNG
In [x]: np.random.randint(1,10, 10)
array([7, 4, 8, 5, 7, 3, 7, 8, 5, 4])    # same as before
```

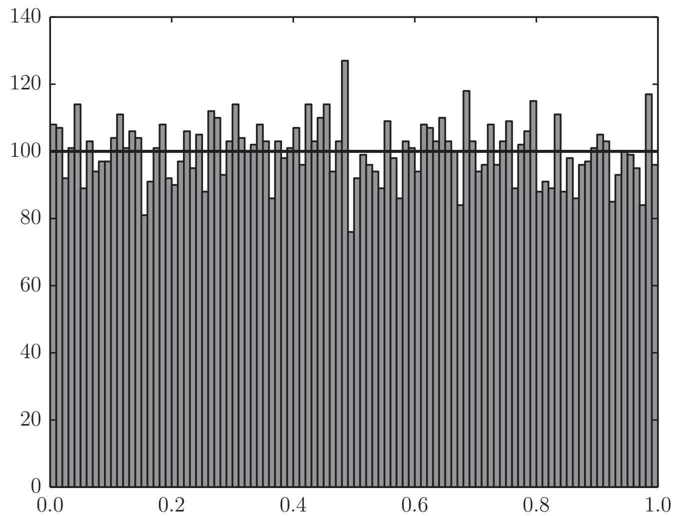


Figure 6.10 Histogram of 10,000 random samples from the uniform distribution on [0, 1) provided by `np.random.random_sample()`.

6.7.1 Uniformly distributed random numbers

Random floating point numbers

The basic random method, `random_sample`[28] takes the shape of an array as its argument and creates an array of the corresponding shape filled with numbers sampled randomly from the uniform distribution over [0, 1); that is, the interval between 0 and 1 inclusive of 0 but exclusive of 1:

```
In [x]: np.random.random_sample((3,2))
array([[ 0.92338355,  0.2978852 ],
       [ 0.75175429,  0.88110707],
       [ 0.16759816,  0.32203783]])
```

(called without an argument, it returns a single random number). If you want numbers sampled from the uniform distribution over $[a, b)$, you need to do a bit of work:

```
In [x]: a, b = 10, 20
In [x]: a + (b-a) * np.random.random_sample((3,2))
array([[ 18.07084068,  12.11591797],
       [ 14.08171741,  19.34857282],
       [ 13.06759203,  11.07003867]])
```

In a uniform distribution, every number has the same probability of being sampled, as can be seen from a histogram of a large number of samples (Figure 6.10):

```
In [x]: pylab.hist(np.random.random_sample(10000), bins=100)
In [x]: pylab.show()
```

The `np.random.rand` method is similar, but is passed the dimensions of the desired array as separate arguments. For example,

[28] `np.random.random_sample` is also available under the aliases `np.random.random`, `np.random.ranf` and `np.random.sample`.

```
In [x]: np.random.rand(2,3)
Out[x]:
array([[ 0.61075227,  0.37459455,  0.95670676],
       [ 0.25276732,  0.1601836 ,  0.3746576 ]])
```

Random integers

Sampling random integers is supported through a couple of methods. The `np.random.randint` method takes up to three arguments `low`, `high` and `size`:

- If both `low` and `high` are supplied, then the random number(s) are sampled from the discrete half-open interval `[low, high)`.[29]
- If `low` is supplied but `high` is not, then the sampled interval is `[0,low)`.
- `size` is the shape of the array of random integers desired. If it is omitted, as with `np.random.rand` a single random integer is returned.

```
In [x]: np.random.randint(4)                # random integer from [0, 4)
2
In [x]: np.random.randint(4, size=10)       # 10 random integers from [0,4)
array([3, 2, 2, 2, 0, 2, 2, 1, 3, 1])
In [x]: np.random.randint(4, size=(3,5))    # array of random ints from [0,4)
array([[0, 1, 1, 2, 2],
       [2, 0, 3, 3, 0],
       [0, 1, 0, 1, 1]])
In [x]: np.random.randint(1, 4, (3,5))      # array of random ints from [1,4)
array([[1, 1, 1, 3, 2],
       [1, 1, 2, 1, 3],
       [1, 3, 1, 3, 1]])
```

`np.random.randint` can be useful for selecting random elements (with replacement) from an array by picking random indexes:

```
In [x]: a = np.array([6,6,6,7,7,7,7,7,7])
In [x]: a[np.random.randint(len(a), size=5)]
array([7, 7, 7, 6, 7])
```

The other method for sampling random integers, `np.random.random_integers` has the same syntax but returns integers sampled from the uniform distribution over the *closed* interval `[low, high]` (if `high` is supplied) or `[0, low]` (if it is not).

Example E6.16 These random integer methods can be used for sampling from a set of evenly spaced real numbers, though it requires a bit of extra work: to pick a number from `n` evenly spaced real numbers between `a` and `b` (inclusive), use

```
In [x]: a + (b-a) * (np.random.random_integers(n) - 1) / (n-1.)
```

For example, to sample from $[\frac{1}{2}, \frac{3}{2}, \frac{5}{2}, \frac{7}{2}]$,

```
In [x]: a, b, n = 0.5, 3.5, 4
In [x]: a + (b-a) * (np.random.random_integers(n, size=10) - 1) / (n-1.)
array([ 1.5,  0.5,  1.5,  1.5,  3.5,  2.5,  3.5,  3.5,  3.5,  3.5])
```

---

[29] Note that this is different from the behavior of the standard library's `random.randint(a,b)` method (see Section 4.5.1) which picks numbers uniformly from the *closed* interval, `[a,b]`.

Example E6.17 In a famous experiment, a group of volunteers are asked to toss a fair coin 100 times and note down the results of each toss (heads, `H`, or tails, `T`). It is generally easy to spot the participants who fake the results by writing down what they think is a random sequence of `H`s and `T`s instead of actually tossing the coin because they tend not to include as many "streaks" of repeated results as would be expected by chance.

If they had access to a Python interpreter, here's how they could produce a more plausibly random set of results:

```
In [x]: res = ['H', 'T']
In [37]: tosses = ''.join([res[i] for i in np.random.randint(2, size=100)])
In [38]: tosses
Out[38]: 'TTHHTHHTTHHHTHTTHHHTHHTHTTHHTHHTTTTHHHHHHHHTTTHTTHHHHHHHTHHHTHHHH
THTTTHTTHHHHTHTTTTHTTTHTHHTTHHHHHHH'
```

This virtual experiment features a run of eight heads in a row, and two runs of seven heads in a row:

```
  TAILS    | i |   HEADS
--------------------------
           | 8 | *
           | 7 | **
           | 6 |
           | 5 |
        ** | 4 | **
       *** | 3 | ***
   ******* | 2 | ******
********** | 1 | ********
```

6.7.2 Random numbers from nonuniform distributions

The full range of random distributions supported by NumPy is described in the official documentation.[30] In the next section we describe in detail only the *normal*, *binomial* and *Poisson* distributions.

The normal distribution

The normal probability distribution is described by the Gaussian function,

$$P(x) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right)$$

where μ is the mean and σ the standard deviation. The NumPy function, `np.random.normal`, selects random samples from the normal distribution. The mean and standard deviation are specified by `loc` and `scale` respectively, which default to 0 and 1. The shape of the returned array is specified with the `size` attribute.

```
In [x]: np.random.normal()
-0.34599057326978105
In [x]: np.random.normal(scale=5., size=3)
```

[30] http://docs.scipy.org/doc/numpy/reference/routines.random.html.

```
array([  4.38196707,  -5.17358738,  11.93523167])
In [x]: np.random.normal(100., 8., size=(4,2))
array([[ 107.730434  ,  101.06221195],
       [ 100.75627505,   88.79995561],
       [  88.82658615,   94.89630767],
       [ 105.91254312,   98.21190741]])
```

It is also possible to draw numbers from the standard normal distribution (that with $\mu = 0$ and $\sigma = 1$) with the `np.random.randn` method. Like `random.rand`, this takes the dimensions of an array as its arguments:

```
In [x]: np.random.randn(2, 2)
array([[-1.25092263,  2.6291925 ],
       [ 0.34158642,  0.40339403]])
```

Although `np.random.randn` does not provide a way to set the mean and standard deviation explicitly, the standard distribution can be rescaled easily enough:

```
In [x]: mu, sigma = 100., 8.
In [x]: mu + sigma * np.random.randn(4, 2)
array([[ 104.92454826,   98.84646729],
       [ 109.43568726,   92.9568489 ],
       [  90.21632016,   96.25271625],
       [ 102.65745451,   89.94890264]])
```

Example E6.18 The normal distribution may be plotted from sampled data as a histogram (Figure 6.11):

```
In [x]: mu, sigma = 100., 8.
In [x]: samples = np.random.normal(loc=mu, scale=sigma, size=10000)
In [x]: counts, bins, patches = pylab.hist(samples, bins=100, normed=True)
In [x]: pylab.plot(bins, 1/(sigma * np.sqrt(2 * np.pi)) *
...                np.exp( -(bins - mu)**2 / (2 * sigma**2) ), lw=2)
In [x]: pylab.show()
```

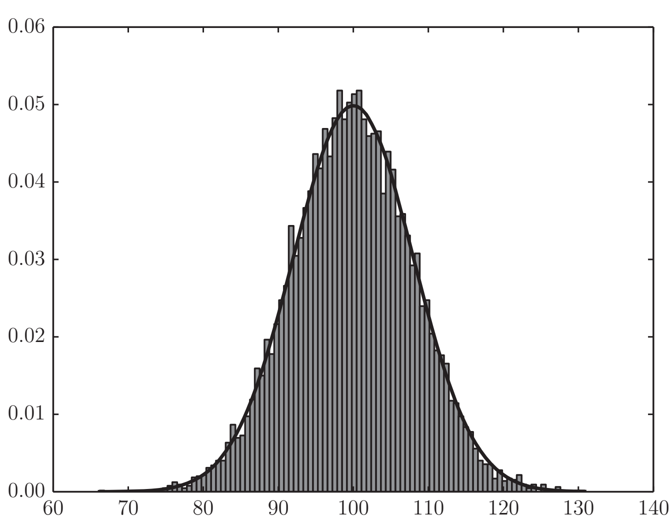


Figure 6.11 Histogram of 10,000 random samples from the normal distribution provided by `np.random.normal`.

6.7.3 The binomial distribution

The binomial probability distribution describes the number of particular outcomes in a sequence of n *Bernoulli trials* – that is, n independent experiments, each of which can yield exactly two possible outcomes (e.g., *yes/no*, *success/failure*, *heads/tails*). If the probability of a single particular outcome (say, *success*) is p, the probability that such a sequence of trials yields exactly k such outcomes is

$$\binom{n}{k}p^k(1-p)^{n-k}, \quad \text{where} \quad \binom{n}{k} = \frac{n!}{k!(n-k)!}.$$

For example, when a fair coin is tossed, the probability of it coming up heads each time is $\frac{1}{2}$. The probability of getting exactly three heads out of four tosses, is therefore $4(\frac{1}{2})(\frac{1}{2})^3 = \frac{1}{4}$, where the factor of $\binom{4}{3} = 4$ accounts for the four possible equivalent outcomes: THHH, HTHH, HHTH, HHHT.

To sample from the binomial distribution described by parameters `n` and `p`, use `np.random.binomial(n, p)`. Again, the shape of an array of samples can be specified with the third argument, `size`:

```
In [x]: np.random.binomial(4, 0.5)
2
In [x]: np.random.binomial(4, 0.5, (4,4))
array([[1, 2, 2, 4],
       [2, 1, 3, 2],
       [2, 3, 1, 1],
       [2, 4, 2, 3]])
```

Example E6.19 There are two stable isotopes of carbon, ^{12}C and ^{13}C (the radioactive ^{14}C nucleus is present in nature in only trace amounts of the order of parts per trillion). Taking the abundance of ^{13}C to be $x = 0.0107$ (i.e., about 1%), we will calculate the relative amounts of buckminsterfullerene, C_{60}, with exactly zero, one, two, three and four ^{13}C atoms. (This is important in nuclear magnetic resonance studies of fullerenes, for example, because only the ^{13}C nucleus is magnetic and so detectable by NMR.)

The number of ^{13}C atoms in a population of carbon atoms sampled at random from a population with natural isotopic abundance follows a binomial distribution: the probability that, out of n atoms, m will be ^{13}C (and therefore $n-m$ will be ^{12}C) is

$$p_m(n) = \binom{n}{m}x^m(1-x)^{n-m}.$$

We can, of course, calculate $p_m(60)$ exactly from this formula for $0 \le m \le 4$, but we can also simulate the sampling with the `np.random.binomial` method:

Listing 6.10 Modeling the distribution of ^{13}C atoms in C_{60}

```
# eg6-e-c13-a.py
import numpy as np

n, x = 60, 0.0107
mmax = 4
m = np.arange(mmax+1)
```

```
# Estimate the abundances by random sampling from the binomial distribution
ntrials = 10000
pbin = np.empty(mmax+1)
for r in m:
❶    pbin[r] = np.sum(np.random.binomial(n, x, ntrials)==r)/ntrials

# Calculate and store the binomial coefficients nCm
nCm = np.empty(mmax + 1)
nCm[0] = 1
for r in m[1:]:
    nCm[r] = nCm[r-1] * (n - r + 1)/ r
# The "exact" answer from binomial distribution
p = nCm * x**m * (1-x)**(n-m)

print('Abundances of C60 as (13C)[m](12C)[60-m]')
print('m   "Exact"    Estimated')
print('-'*24)
for r in m:
    print('{:1d}   {:6.4f}      {:6.4f}'.format(r, p[r], pbin[r]))
```

❶ For each value of `r` in the array `m`, we sample a large number of times (`ntrial`) from the binomial distribution described by $n = 60$ and probability, $x = 0.0107$. The comparison of these sample values with a given value of `r` yields a boolean array which can be summed (remembering that `True` evaluates to 1 and `False` evaluates to 0); division by `ntrials` then gives an estimate of the probability of exactly `r` atoms being of type ^{13}C and the remainder of type ^{12}C.

The explicit loop over `m` could be removed by creating an array of shape `(ntrials, mmax+1)` containing all the samples, and summing over the first axis of this array in the comparison with the `m` array:

```
samples = np.random.binomial(n, x, (ntrials, mmax+1))
pbin = np.sum(samples == m, axis=0) / ntrials
```

The abundances of $^{13}\mathrm{C}_\mathrm{m}^{12}\mathrm{C}_{60-\mathrm{m}}$ produced by our program are given as the following output.

```
Abundances of C60 as (13C)[m](12C)[60-m]
m   "Exact"    Estimated
------------------------
0   0.5244      0.5199
1   0.3403      0.3348
2   0.1086      0.1093
3   0.0227      0.0231
4   0.0035      0.0031
```

That is, almost 48% of C_{60} molecules contain at least one magnetic nucleus.

6.7.4 The Poisson distribution

The Poisson distribution describes the probability of a particular number of independent events occurring in a given interval of time if these events occur at a known average rate. It is also used for occurrences in specified intervals over other domains such as distance

or volume. The Poison probability distribution of the number of events, k, is

$$P(k) = \frac{\lambda^k e^{-\lambda}}{k!},$$

where the parameter λ is the expected (average) number of events occurring within the considered interval.[31] The NumPy implementation `np.random.poisson` takes λ as its first argument (which defaults to 1) and, as before the shape of the desired array of samples can be specified with a second argument, `size`. For example, if I receive an average of 2.5 emails an hour, a sample of the number of emails I receive each hour over the next 8 hours could be obtained as:

```
In [x]: np.random.poisson(2.5, 8)
array([4, 1, 3, 0, 4, 1, 3, 2])
```

Example E6.20 The endonuclease enzyme *Eco*RI is used as a restriction enzyme which cuts DNA at the nucleic acid sequence GAATTC. Suppose a given DNA molecule contains 12000 base pairs and a 50% G+C content. The Poisson distribution can be used to predict the probability that *Eco*RI will fail to cleave this molecule as follows:

The recognition site, GAATTC, consists of six nucleotide base pairs; the probability that any given six-base sequence corresponds to GAATTC is $1/4^6 = 1/4096$ and so the expected number of cleavage sites for *Eco*RI in this DNA molecule is $\lambda = 12000/4096 = 2.93$. From the Poisson distribution, we expect the probability that the endonuclease will fail to cleave this molecule is therefore

$$P(0) = \frac{\lambda^0 e^{-\lambda}}{0!} = 0.053,$$

or about 5.3%. To simulate the possibilities stochastically:

```
In [x]: lam = 12000 / 4**6
In [x]: N = 100000
In [x]: np.sum(np.random.poisson(lam, N)==0)/N
Out[x]: 0.053699999999999998
```

6.7.5 Random selections, shuffling and permutations

It is often the case that given an array of values, you wish to pick one or more at random (with or without replacement). This is the purpose of the `np.random.choice` method. Given a single argument, an one-dimensional sequence, it returns a random element drawn from the sequence:

```
In [x]: np.random.choice([ 1,  5,  2, -5,  5,  2,  0])
2
In [x]: np.random.choice(np.arange(10))
7
```

[31] The Poisson distribution is the limit of the binomial distribution as $n \to \infty$ and $p \to 0$ such that $\lambda = np$ tends to some finite constant value.

A second argument, `size`, controls the shape of the array of random samples returned, as before. By default, the elements of the sequence are drawn randomly with a uniform distribution and *with* replacement; to draw the sample *without* replacement, set `replace=False`.

```
In [x]: a = np.array([1, 2, 0, -1, 1])
In [x]: np.random.choice(a, 6)  # six random selections from a
array([ 1, -1,  2,  1, -1,  1])
In [x]: np.random.choice(a, (2,2), replace=False)
array([[ 2, -1],
       [ 1,  0]])
In [x]: np.random.choice(a, (3,2), replace=False)
... <some traceback information> ...
ValueError: Cannot take a larger sample than population when 'replace=False'
```

This last example shows that, as you might expect, it is not possible to draw a larger number of elements than there are in the original population if you are sampling without replacement.

To specify the probability of each element being selected, pass a sequence of the same length as the population to be sampled as the argument `p`. The probabilities should sum to 1.

```
In [x]: a = np.array([1, 2, 0, -1, 1])
In [x]: np.random.choice(a, 5, p=[0.1, 0.1, 0., 0.7, 0.1])
Out[x]: array([-1, -1, -1, -1,  1])
In [x]: np.random.choice(a, 2, False, p=[0.1, 0.1, 0., 0.8, 0.])
Out[x]: array([-1,  2])         # sample without replacement
```

There are two methods for permuting the contents of an array: `np.random.shuffle` randomly rearranges the order of the elements *in place* whereas `np.random.permutation` makes a copy of the array first, leaving the original unchanged:

```
In [x]: a = np.arange(6)
In [x]: np.random.permutation(a)
array([4, 2, 5, 1, 3, 0])
In [x]: a
array([0, 1, 2, 3, 4, 5])
In [x]: np.random.shuffle(a)
In [x]: a
array([5, 4, 1, 3, 0, 2])
```

These methods only act on the first dimension of the array:

```
In [x]: a = np.arange(6).reshape(3, 2)
In [x]: a
array([[0, 1],
       [2, 3],
       [4, 5]])
In [x]: a.random.permutation(a) # permutes the rows, but not the columns
array([[2, 3],
       [4, 5],
       [0, 1]])
```

6.7.6 Exercises

Questions

Q6.7.1 Explain the difference between

```
In [x]: a = np.array([6,6,6,7,7,7,7,7,7])
In [x]: a[np.random.randint(len(a), size=5)]
array([7, 7, 7, 6, 7])      # (for example)
```

and

```
In [x]: np.random.randint(6, 8, 5)
array([6, 6, 7, 7, 7])      # (for example)
```

Q6.7.2 In Example E6.16 we used `random.random_integers` to sample from the uniform distribution on the floating point numbers $[\frac{1}{2}, \frac{3}{2}, \frac{5}{2}, \frac{7}{2}]$. How can you do the same using the `random.randint` instead?

Q6.7.3 The American lottery, Mega Millions, at the time of writing, involves the selection of five numbers out of 75 *and* one from 15. The jackpot is shared among the players who match all of their numbers in a corresponding random draw. What is the probability of winning the jackpot? Write a single line of Python code using NumPy to pick a set of random numbers for a player.

Q6.7.4 Suppose an n-page book is known to contain m misprints. If the misprints are independent of one another, the probability of a misprint occurring on a particular page is $p = 1/n$ and their distribution may be considered to be binomial. Write a short program to conduct a number of trial virtual "printings" of a book with $n = 500, m = 400$ and determine the probability, Pr, that a single given page will contain two or more misprints.

Compare with the result predicted by the Poisson distribution with rate parameter $\lambda = m/n$, $\mathrm{Pr} = 1 - e^{-\lambda}\left(\frac{\lambda^0}{0!} + \frac{\lambda}{1!}\right)$.

Problems

P6.7.1 Simulate an experiment carried out `ntrials` times in which, for each experiment, `n` coins are tossed and the total number of heads each time is recorded.

Plot the results of the simulation on a suitable histogram and compare with the expected binomial distribution of heads.

P6.7.2 A classic problem, first posed by Georges-Louis Leclerc, Comte de Buffon, can be stated as follows:

> Given a plane ruled with parallel lines a distance d apart, what is the probability that a needle of length $l \leq d$ dropped at random onto the plane will cross a line?

The problem can be solved analytically, yielding the answer $2l/\pi d$; show that this solution is given approximately for the case $l = d$ using a random simulation (Monte Carlo) method, that is, by simulating the experiment with a large number of random orientations of the needle.

A related problem involves dropping a circular coin of radius a onto a floor consisting of square tiles, each of side d. Show that the probability of a coin crossing a tile edge is $1-(d-2a)^2/d^2$ and confirm it with a Monte Carlo simulation.

P6.7.3 Some bacteria, such as *E. coli*, possess helical flagella which enable them to move toward attractants such as nutrients, process known as chemotaxis. When the flagella rotate counterclockwise the bactrium is propelled forward; when they rotate clockwise, it tumbles randomly, changing its orientation. A combination of such movements enables the bacterium to perform a *biased random walk*: if the bacterium senses it is moving up a concentration gradient toward an attractant it will rotate its flagella counterclockwise more often than clockwise so as to continue moving in that direction; conversely, if it is moving away it is more likely to rotate its flagella clockwise so as to tumble with the aim of randomly changing its orientation to one that points it toward the attractant.

The chemotaxis of *E. coli* may be modeled (very) simplistically by considering a bacterium to move in a two-dimensional "world" populated by an attractant with a constant concentration gradient away from some location. At each of a series of time steps, a model bacterium detects whether it is moving up or down this gradient and either continues moving or tumbles according to some pair of probabilities.

Write a Python program to implement this simple model of chemotaxis for a world consisting of the unit square with an attractant at its center. Plot the locations of 10 model bacteria that start off evenly spaced around the unit circle centered on the attractant location.

P6.7.4 One way to simulate the meanders in a river is as the average of a large number of a random walks.[32] Using a coordinate system (x, y), start at point $A = (0, 0)$ and aim to finish at $B = (b, 0)$. Starting from an initial heading of ϕ_0 from the AB direction, at each step change this angle by a random amount drawn from a normal distribition with mean $\mu = 0$ and standard deviation σ, and proceed by unit distance in this direction. Discard any walks which do not, after n steps, finish within one unit of B (this will be the majority!).

Write a program to find the average path meeting the above constraints for $b = 10$, using $\phi_0 = 110°$, $\sigma = 17°$, $n = 40$ and 10^6 random walk trials. Plot the accepted walks and their average, which should resemble a meander.

6.8 Discrete Fourier transforms

6.8.1 One-dimensional Fast Fourier Transforms

`numpy.fft` is NumPy's Fast Fourier Transform (FFT) library for calculating the discrete Fourier transform (DFT) using the ubiquitous Cooley and Tukey algorithm.[33] The

[32] B. Hayes, (2006) *American Scientist* **94**, 490; H. von Schelling, *General Electric Report* No. 64GL92

[33] J. W. Cooley and J. W. Tukey, (1965) *Math. Comput.* **19**, 297=301.

definition for the DFT of a function defined on n points, $f_m, m = 1, 2, \cdots, n-1$ used by NumPy is

$$F_k = \sum_{m=0}^{n-1} f_m \exp\left(-\frac{2\pi imk}{n}\right), \quad k = 0, 1, 2, \cdots, n-1 \tag{6.1}$$

NumPy's basic DFT method, for real and complex functions, is `np.fft.fft`. If the input signal function, f, is considered to be in the time domain, the output Fourier Transform, F, is in the frequency domain and is returned by the `fft(f)` function call in a standard order: `F[:n/2]` are the positive-frequency terms in increasing order, `F[n/2+1:]` contains the negative-frequency terms in decreasing order, and `F[n/2]` is the (positive and negative) Nyquist frequency.[34] `np.abs(F)`, `np.abs(F)**2` and `np.angle(F)` are the *amplitude spectrum*, *power spectrum* and *phase spectrum* respectively.

The frequency bins corresponding to the values of `F` are given by `np.fft.fftfreq (n, d)` where `d` is the sample spacing. For even n, this is equivalent to

$$0, \frac{1}{dn}, \frac{2}{dn}, \cdots, \frac{n/2-1}{dn}, -\frac{n/2}{dn}, -\frac{n/2-1}{dn}, \cdots, -2, -1$$

To shift the spectrum so that the zero-frequency component is at the center, call `np.fft.fftshift`. To undo that shift, call `np.fft.ifftshift`.

For example, consider the following waveform in the time domain with some synthetic Gaussian noise added:

$$f(t) = 2\sin(20\pi t) + \sin(100\pi t)\,.$$

```
In [x]: A1, A2 = 2, 1
In [x]: freq1,freq2 =  10, 50
In [x]: fsamp = 500
In [x]: t = np.arange(0, 1, 1/fsamp)
In [x]: n = len(t)
In [x]: f = A1*np.sin(2*np.pi*freq1*t) + A2*np.sin(2*np.pi*freq2*t)
In [x]: f += 0.2 * np.random.randn(n)
In [x]: pylab.plot(t, f)
In [x]: pylab.xlabel('Time /s')
In [x]: pylab.show()
```

The plot of this waveform is depicted in Figure 6.12.

The Fourier transform of this function is complex; its real and imaginary components are plotted here (Figure 6.13).

```
In [x]: F = np.fft.fft(f)
In [x]: pylab.plot(F.real, 'k', label='real')
In [x]: pylab.plot(F.imag, 'gray', label='imag')
In [x]: pylab.legend(loc=2)
In [x]: pylab.show()
```

[34] Here, `n` is assumed to be even.

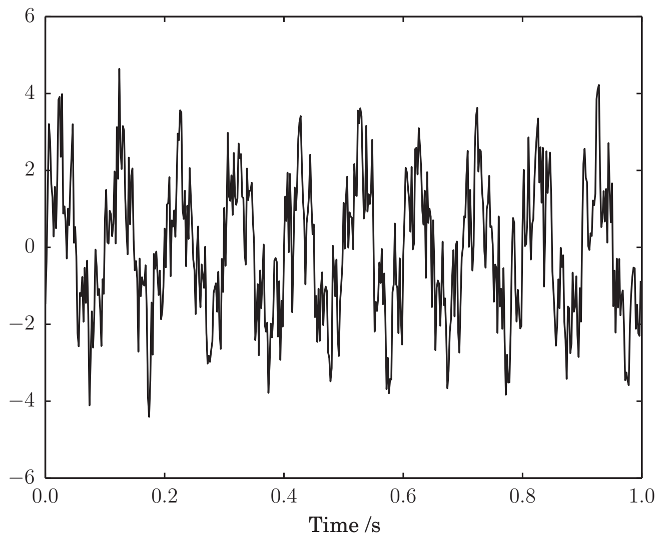


Figure 6.12 The noisy waveform referred to in the text.

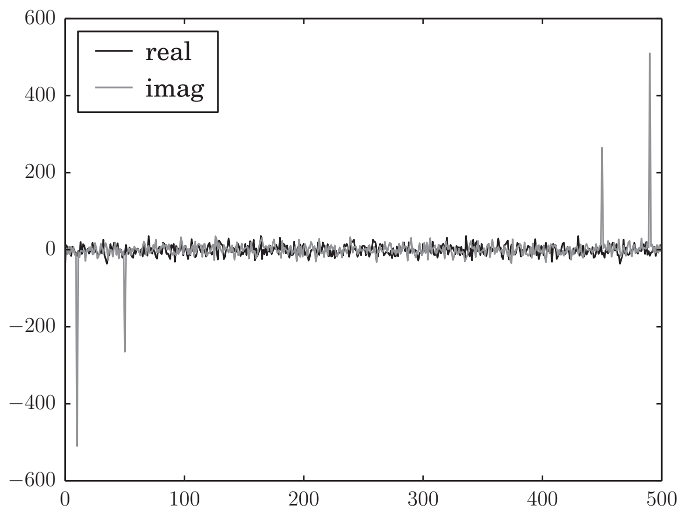


Figure 6.13 The Fourier transform of a noisy waveform with two frequency components, as returned by `np.fft.fft`.

Now look at the shifted amplitude spectrum with the zero-frequency component at the center:[35]

```
In [x]: freq = np.fft.fftfreq(n, 1/fsamp)
In [x]: F_shifted = np.fft.fftshift(F)
In [x]: freq_shifted = np.fft.fftshift(freq)
In [x]: pylab.plot(freq_shifted, np.abs(F_shifted))
In [x]: pylab.xlabel('Frequency /Hz')
In [x]: pylab.show()
```

This plot is given in Figure 6.14.

[35] The shifting here is for illustration: note that it isn't really necessary to shift both `freq` and `F` arrays simply to plot one against the other.

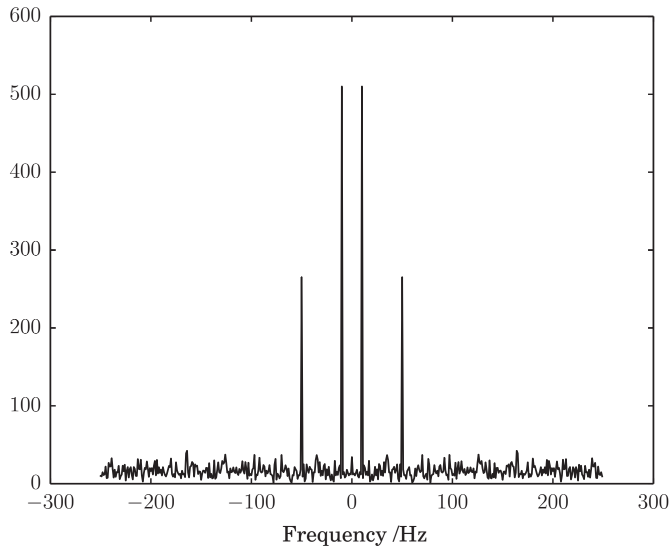


Figure 6.14 The Fourier Transform of a noisy waveform with two frequency components plotted against frequency.

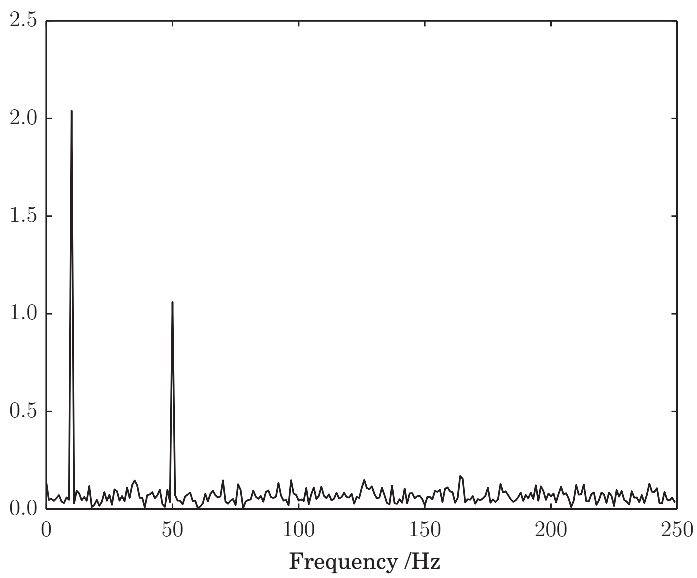


Figure 6.15 The positive-frequency components of the Fourier transform of a noisy waveform, normalized to show their intensities.

Now, because our input function is real, its Fourier transform is Hermitian: the negative frequency components are the complex conjugates of the positive frequency components so they don't contain any further information. Therefore, we only need to deal with the first half of the `F` array. Plotted against its (positive) frequencies as an amplitude spectrum (Figure 6.15):

❶
```
In [x]: spec = 2/n * np.abs(F[:n/2])
In [x]: pylab.plot(freq[:n/2], spec, 'k')
In [x]: pylab.xlabel('Frequency /Hz')
In [x]: pylab.show()
```

❶ Note that because of the way this DFT has been defined, a normalization factor of $\frac{2}{n}$ is required to faithfully regenerate the original amplitudes of each component.

The amplitudes of the 10 Hz and 50 Hz signals are easily resolved in this spectrum.

The inverse Fourier Transform defined through

$$f_m = \frac{1}{n}\sum_{k=0}^{n-1} F_k \exp\left(\frac{2\pi imk}{n}\right) \qquad m = 0, 1, 2, \cdots, n-1$$

is returned by the method `np.fft.ifft`.

If, as mentioned earlier, the input function array is real and only the non-negative frequency components are needed, the `np.fft` methods `rfft`, `irfft`, `rfftfreq` can be used.

6.8.2 Two-dimensional Fast Fourier Transforms

Discrete Fourier transforms and their inverses in two and higher dimensions are possible using the `np.fft` methods `fft2`, `ifft2`, `fftn` and `ifftn`. The two-dimensional DFT is defined as

$$F_{jk} = \sum_{p=0}^{m-1}\sum_{q=0}^{n-1} f_{pq} \exp\left[-2\pi i\left(\frac{pj}{m} + \frac{qk}{n}\right)\right],$$
$$j = 0, 1, 2, \cdots, m-1;\; k = 0, 1, 2, \cdots, n-1.$$

and higher dimensions follow similarly.

Example E6.21 The two-dimensional DFT is widely used in image processing.[36] For example, multiplying the DFT of an image by a two-dimensional Gaussian function is a common way to blur an image by decreasing the magnitude of its high-frequency components.

The following code produces an image of randomly arranged squares and then blurs it with a Gaussian filter.

Listing 6.11 Blurring an image with a Gaussian filter

```
# eg6-fft2-blur.py
import numpy as np
import pylab

# image size, square side length, number of squares
ncols, nrows = 120, 120
sq_size, nsq = 10, 20

# The image array (0=background, 1=square) and boolean array of allowed places
# to add a square so that it doesn't touch another or the image sides
```

[36] Note that there is an entire SciPy subpackage, `scipy.ndimage`, not described in this book, devoted to image processing. This example serves simply to illustrate the syntax and format of NumPy's two-dimensional FFT implementation.

```
image = np.zeros((nrows, ncols))
sq_locs = np.zeros((nrows, ncols), dtype=bool)
sq_locs[1:-sq_size-1:,1:-sq_size-1] = True

def place_square():
    """ Place a square at random on the image and update sq_locs. """
    # valid_locs is an array of the indexes of True entries in sq_locs
    valid_locs = np.transpose(np.nonzero(sq_locs))
    # pick one such entry at random, and add the square so its top left
    # corner is there; then update sq_locs
    i, j = valid_locs[np.random.randint(len(valid_locs))]
    image[i:i+sq_size, j:j+sq_size] = 1
    imin, jmin = max(0,i-sq_size-1), max(0, j-sq_size-1)
    sq_locs[imin:i+sq_size+1, jmin:j+sq_size+1] = False

# Add the required number of squares to the image
for i in range(nsq):
    place_square()
pylab.imshow(image)
pylab.show()

# Take the two-dimensional DFT and center the frequencies
ftimage = np.fft.fft2(image)
ftimage = np.fft.fftshift(ftimage)
pylab.imshow(np.abs(ftimage))
pylab.show()


# Build and apply a Gaussian filter.
sigmax, sigmay = 10, 10
cy, cx = nrows/2, ncols/2
x = np.linspace(0, nrows, nrows)
y = np.linspace(0, ncols, ncols)
X, Y = np.meshgrid(x, y)
gmask = np.exp(-(((X-cx)/sigmax)**2 + ((Y-cy)/sigmay)**2))

ftimagep = ftimage * gmask
pylab.imshow(np.abs(ftimagep))
pylab.show()

# Finally, take the inverse transform and show the blurred image
imagep = np.fft.ifft2(ftimagep)
pylab.imshow(np.abs(imagep))
pylab.show()
```

The results are shown in Figure 6.16.

6.8.3 Exercises

Questions

Q6.8.1 Compare the speed of execution of NumPy's `np.fft.fft` algorithm and that of the direct implementation of Equation 6.1.

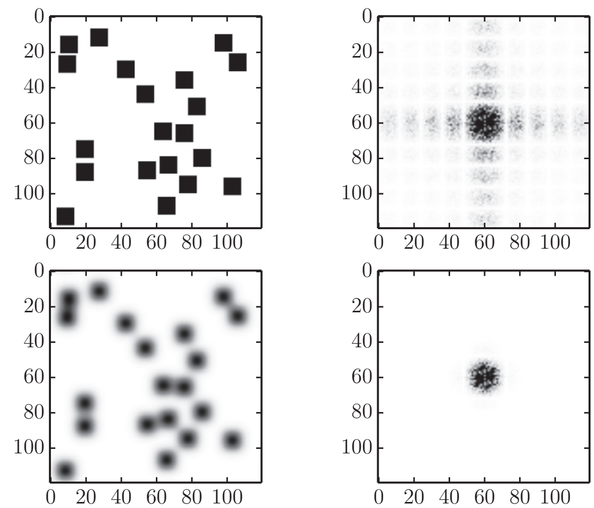


Figure 6.16 Blurring an image with a Gaussian filter applied its two-dimensional Fourier transform.

Hint: treat the direct equation as a matrix multiplication (dot product) of an array of n function values (random ones will do) with the $n \times n$ array with entries $\exp(-2\pi imk/n)$ ($m, k = 0, 1, \cdots, n-1$). Use IPython's `%timeit` magic.

Problems

P6.8.1 Consider a signal in the time domain defined by the function

$$f(t) = \cos(2\pi\nu t)e^{-t/\tau},$$

with frequency $\nu = 250$ Hz decaying exponentially with a lifetime $\tau = 0.2$ s. Plot the function, sampled at 1,000 Hz, and its discrete Fourier transform against frequency. Examine, by means of a suitable plot, the effect of *apodization* on the DFT by truncating the time sequence after (a) 0.5 s, (b) 0.2 s.

P6.8.2 A square wave of period T may be defined through the following function:

$$f_{\mathrm{sq}}(t) = \begin{cases} 1 & t < T/2 \\ -1 & t \geq T/2 \end{cases}$$

with $f(t) = f(t + nT)$ for $n = \pm 1, \pm 2, \cdots$.

Plot the square wave with $T = 1$ (and hence cycle frequency, $\nu = 1$) for $0 \leq t < 2$ taking a grid of 2,048 time points over this interval. Calculate and plot its discrete Fourier transform.

The Fourier *expansion* of this function is the infinite series

$$f_{\mathrm{sq}}(t) = \frac{4}{\pi}\sum_{k=1}^{\infty}\frac{1}{2k-1}\sin[2\pi(2k-1)\nu t]$$

Compare the square wave function with this Fourier expansion truncated at 3, 9 and 18 terms. Also compare their (suitably normalized) Fourier transforms: the missing

frequencies in each truncated series should appear as zeros in its Fourier transform, whereas the present terms will have intensities $4/[\pi(2k-1)]$.

P6.8.3 The `scipy` library provides a routine for reading in `.wav` files as NumPy arrays:

```
In [x]: from scipy.io import wavfile
In [x]: sample_rate, wav = wavfile.read(\emph{<filename>})
```

For a stereo file, the array `wav` has shape `(n,2)` where `n` is the number of samples.

Use the routines of `np.fft` to identify the chords present in the sound file `chords.wav`, which may be downloaded from scipython.com/ex/afi. Which major chord do they comprise?

The frequencies of musical notes on an equal-tempered scale for which $A_4 = 440$ Hz are provided as a dictionary in the file `notes.py`.

7 Matplotlib

Matplotlib is probably the most popular Python package for plotting data. It can be used through the procedural interface `pylab` in very quick scripts to produce simple visualizations of data (see Chapter 3) but, as described in this chapter, with care it can also produce high-quality figures for journal articles, books and other publications. Although there is some limited functionality for producing three-dimensional plots (see Section 7.2.3), it is primarily a two-dimensional plotting library.

7.1 Matplotlib basics

Matplotlib is a large package organized in a hierarchy: at the highest level is the `matplotlib.pyplot` module. This provides a "state-machine environment" with a similar interface to MATLAB and allows the user to add plot elements (data points, lines, annotations, etc.) through simple function calls. This is the interface used by `pylab`, which was introduced in Chapter 3.

At a lower level, which allows more advanced and customizable use, Matplotlib has an object-oriented interface that allows one to create a *figure* object to which one or more *axes* objects are attached. Most plotting, annotation and customization then occurs through these axes objects. This is this approach we adopt in this chapter.

To use Matplotlib in this way, we use the following recommended imports:

```
import matplotlib.pyplot as plt
import numpy as np
```

7.1.1 Basic figures

Plotting on a single axes object

The top-level object, containing all the elements of a plot is called `Figure`. To create a figure object, call `plt.figure`. No arguments are necessary, but optional customization can be specified by setting the values described in Table 7.1. For example,

```
In [x]: # a default figure, with title "Figure 1"
In [x]: fig = plt.figure()

In [x]: # a small figure with red background
In [x]: fig = plt.figure('Population density', figsize=(4.5, 2.),
  ....:                  facecolor='red')
```

Table 7.1 Arguments to `plt.figure`

Argument	Description
`num`	An identifier for the figure – if none is provided, an integer, starting at 1, is used and incremented with each figure created. Alternatively using a string will set the window title to that string when the figure is displayed with `plt.show()`.
`figsize`	A tuple of figure `(width, height)`, unfortunately in inches.
`dpi`	Figure resolution in dots-per-inch.
`facecolor`	Figure background color.
`edgecolor`	Figure border color.

To actually plot data, we need to create an `Axes` object – a region of the figure containing the axes, tick-marks, labels, plot lines and markers, and so on. The simplest figure, consisting of a single `Axes` object, is created and returned with

```
In [x]: ax = fig.add_subplot(111)
```

The argument `111` here is a commonly used abbreviation for the tuple `(1,1,1)` specifying subplot `1` of a figure with `1` row and `1` column of subplots (see Section 7.1.3). The `Axes` object, `ax`, is the one on which we can actually plot the data with `ax.plot`. The essential features of this `plot` method were described in Chapter 3. Here, however, we note that the `plot` method actually returns a `list` of objects representing the plotted lines. In its simplest usage, only a single line is plotted, and so this list consists of one `Line2D` object that we may assign to a variable if desired. As a full example, consider the following comparison of the catenary $y = \cosh(x)$ and its parabolic approximation, $y = 1 + x^2/2$.

```
import matplotlib.pyplot as plt
import numpy as np

fig = plt.figure()
ax = fig.add_subplot(111)

x = np.linspace(-2, 2, 1000)
line_cosh, = ax.plot(x, np.cosh(x))   ❶
line_quad, = ax.plot(x, x**2 / 2)

plt.show()
```

❶ Note the syntax `line_cosh, = ...` to assign the returned line object to the variable `line_cosh` rather than the `list` containing that object.

The two plotted lines are shown in Figure 7.1.

Plot limits

By default, Matplotlib plots all of the data passed to `plot` and sets the axis limits accordingly. To set the axis limits to something else, use the `ax.set_xlim` and `ax.set_ylim` methods. Either both limits can be set or an individual limit can be set with the arguments `left`, `right` (or `xmin`, `xmax`) and `bottom`, `top` (or `ymin`, `ymax`). Unspecified limits are left unchanged. For example,

Table 7.2 Matplotlib line styles

	(no line)
`-`	solid
`--`	dashed
`:`	dotted
`-.`	dash-dot

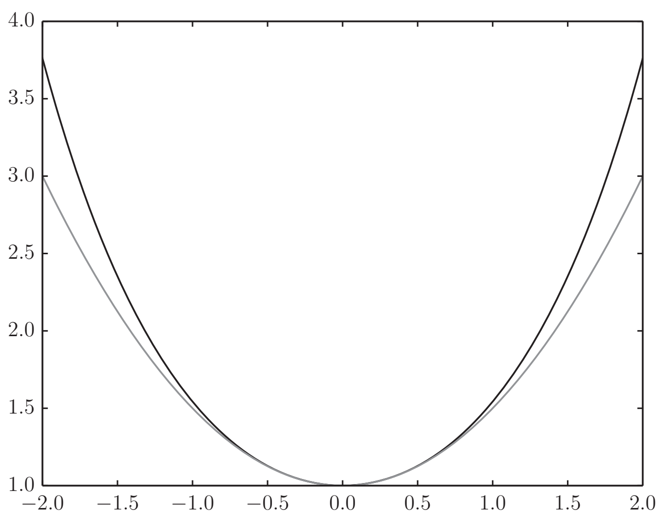


Figure 7.1 A simple plot of two lines on a single `Axes` object.

```
x = np.linspace(-3,3,1000)
y = x**3 + 2 * x**2 - x - 1
fig = plt.figure()
ax = fig.add_subplot(111)
ax.plot(x,y)

ax.set_xlim(-1,2)        # x-limits are -1 to 2
ax.set_ylim(bottom=0)    # ymin=0: plot will be "clipped" at the bottom
```

If `bottom` is greater than `top` or `right` less than `left`, the corresponding axis will be reversed; that is, values on this axis will *decrease* from left to right (or from bottom to top) (see Exercise P7.1.5).

If you wish to invert the axis direction without changing the limit values, the method calls `ax.invert_xaxis()` and `ax.invert_yaxis()` will do that for you.

Line styles, markers and colors

As with `pylab`, the plot style can be specified by passing extra arguments to the `plot()` method. The default line style is a solid, 1.0 pt weight line in a color determined by the order in which it is added to the plot.

An alternative line style can be selected from the predefined options with the `linestyle` (or simply `ls`) argument. Possible string values to pass to this argument (including the empty string for plotting no line) are shown in Table 7.2.

Table 7.3 Matplotlib colour code letters

b	blue
g	green
r	red
c	cyan
m	magenta
y	yellow
k	black
w	white

Further customization is possible by setting the `dashes` argument to a sequence of values describing the repeated dash pattern in points. For example, `dashes=[2, 4, 8, 4, 2, 4]]` represents a pattern of dot (2 pts), space (4 pts), dash (8 pts), space (4 pts), dot (2 pts), space (4 pts) to be repeated as the line style. Equivalently, one can call a plotted line's `set_dashes` method, as in the following code snippet:

```
x = np.linspace(-np.pi, np.pi, 1000)
line, = plt.plot(x, np.sin(x))
line.set_dashes([2, 4, 8, 4, 2, 4])      # dot-dash-dot
```

The line weight is customized by setting the `lineweight` (or simply `lw`) argument to a number of points.

Line colors are specified with the `color` (or simply `c`) argument used in one of several ways:

- *string*: by letter or name, one of the values given in Table 7.3.
- *string*: by HTML 6-digit hex-string preceded by `'#'`, for example `'#ffff00'` is yellow.
- *string*: a string representation of a `float` between 0. and 1. (for example `'0.4'`) gives a gray-scale between black (0.) and white(1.).
- *tuple of floats* between 0. and 1.: RGB components, for example `(0.5, 0., 0.)` is a dark red color.

By default, the `Line2D` object created by calling `plot` on an `Axes` object does not include *markers*: symbols printed at each point on the plot. To add them, specify one of the single-character marker codes given in Table 7.4 using the `marker` argument

```
ax.plot(x, y, marker='v')    # downward pointing triangles
```

Other marker properties can be set with the arguments listed in Table 7.5.

Matplotlib markers can be further customized; see the documentation for details.[1]

Scatterplots

A typical two-dimensional scatterplot depicts the data as points on a Cartesian axes system. Sometimes there is no meaningful or helpful ordering to the data and so no

[1] http://matplotlib.org/api/markers_api.html.

Table 7.4 Some Matplotlib marker styles (single character sting codes)

Code	Marker	Description
`.`	·	point
`o`	○	circle
`+`	+	plus
`x`	×	x
`D`	◇	diamond
`v`	▽	(downward triangle)
`^`	△	(upward triangle)
`s`	□	square
`*`	⋆	star

Table 7.5 Matplotlib marker properties

Argument	Abbreviation	Description
`markersize`	`ms`	Marker size, in points
`markevery`		Set to a positive integer, *N*, to print a marker every *N* points; the default, `None`, prints a marker for every point
`markerfacecolor`	`mfc`	Fill color of the marker
`markeredgecolor`	`mec`	Edge color of the marker
`markeredgewidth`	`mew`	Edge width of the marker, in points

need to join data points by lines. The `pyplot.scatter` function creates a scatterplot. In addition to one-dimensional sequences of $x-$ and $y-$ data, as for `pyplot.plot`, the data point marker colors and sizes can be set individually by passing a sequence of appropriate values of the same length as the data to the arguments `s` and `c` respectively. The marker sizes are in points^2 (*points squared*) so that their *area* is proportional to the values passed to `s`. Manipulating the size of the markers is a common way of indicating a third dimension to the data, as in the following example.

Example E7.1 To explore the correlation between birth rate, life expectancy and per capita income, we may use a scatterplot. Note that the marker sizes are set in proportion to the countries' percapita GDP but have to be scaled a little so they don't get too large (see Figure 7.2).

Listing 7.1 Scatterplot of demographic data for eight countries

```
# eg7-scatter.py

import numpy as np
import matplotlib.pyplot as plt

countries = ['Brazil', 'Madagascar', 'S. Korea', 'United States',
             'Ethiopia', 'Pakistan', 'China', 'Belize']
# Birth rate per 1000 population
```

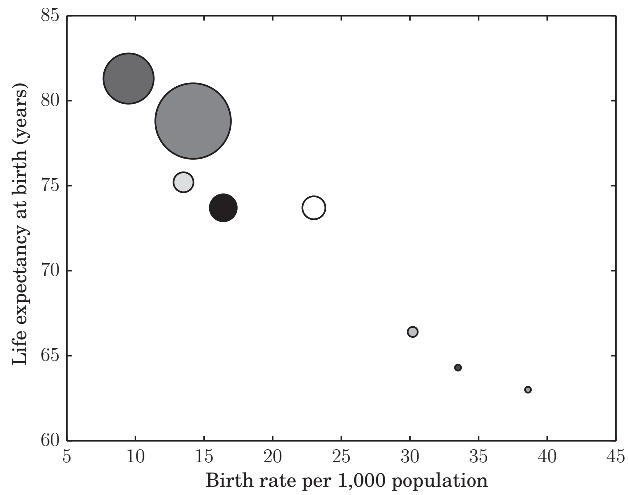


Figure 7.2 A scatterplot with variable marker sizes indicating each country's GDP.

```
birth_rate = [16.4, 33.5, 9.5, 14.2, 38.6, 30.2, 13.5, 23.0]
# Life expectancy at birth, years
life_expectancy = [73.7, 64.3, 81.3, 78.8, 63.0, 66.4, 75.2, 73.7]
# Per person income fixed to US Dollars in 2000
GDP = np.array([4800, 240, 16700, 37700, 230, 670, 2640, 3490])

fig = plt.figure()
ax = fig.add_subplot(111)
# Some arbitrary colors:
colors = range(len(countries))
ax.scatter(birth_rate, life_expectancy, c=colors, s=GDP/10)
ax.set_xlabel('Birth rate per 1000 population')
ax.set_ylabel('Life expectancy at birth (years)')

plt.show()
```

Gridlines

Gridlines are vertical (for the *x*-axis) and horizontal (for the *y*-axis) lines running across the plot to aid with locating the numerical values of data points. By default no gridlines are drawn, but they may be turned on by calling `grid` method on an `Axes` object (to add both horizontal and vertical gridlines) or the `xaxis` or `yaxis` objects of a given `Axes` (to select the gridlines to use). For example,

```
ax.yaxis.grid(True)   # Turn on horizontal gridlines
```

or

```
ax.grid(True)    # Turn on all gridlines
```

The line properties of the gridlines are set with the `linestyle`, `linewidth`, `color`, etc. arguments as for plot lines.

Two sorts of gridlines correspond to the major and minor tick marks (see below): these can be selected with the `which` argument, which takes the values `'major'`, `'minor'` and `'both'`. The default (if not specified) is `which='major'`.

```
ax.xaxis.grid(True, which='minor', c='b')    # Minor x-axis gridlines in blue
```

Log scales

By default, Matplotlib plots data on a linear scale. To set a logarithmic scale, call one or both of the following on your `Axes` object:

```
ax.set_xscale('log')
ax.set_yscale('log')
```

Base-10 logarithms are used by default, but the (integer) base can be set with the optional arguments `basex` or `basey`. Nonpositive values in the data will be masked as invalid by default. If you want negative values to be handled "symmetrically" with positive ones, such that $\log(-|x|) = -\log(|x|)$, then use `'symlog'` instead of `'log'`. See also Question 7.1.1.

Adding titles, labels and legends

Axis labels may be added to the subplot `Axes` object with `ax.set_xlabel` and `ax.set_ylabel`.

Plot line legend labels are defined by adding the `label` attribute to the `plt.plot` function call. However, the legend itself will not appear unless `legend` is called on the plot `Axes` object (e.g., with `ax.legend()`.) The appearance of the legend itself can be customized extensively, but the most common additional argument you may wish to pass to is `legend` is `loc`, defining the location of the legend on the plot (see Table 3.1).

There are two types of title you may want to give your figure: `fig.suptitle` adds a centered title to the entire figure, which may contain more than one subplot; `ax.title` adds a title to a single subplot.[2]

Example E7.2 The data read in from the file `eg7-marriage-ages.txt`, which can be downloaded from scipython.com/eg/aag, giving the median age at first marriage in the United States for 13 decades since 1890 are plotted by the program below. Grid lines are turned on for both axes with `ax.grid()`, and custom markers are used for the data points themselves (see Figure 7.3).

Listing 7.2 The median age at first marriage in the US over time

```
# eg7-marriage-ages.py
import numpy as np
import matplotlib.pyplot as plt

year, age_m, age_f = np.loadtxt('eg7-marriage-ages.txt', unpack=True,skiprows=3)
fig = plt.figure()
ax = fig.add_subplot(111)
```

[2] See the documentation at http://matplotlib.org/api/legend_api.html for more details.

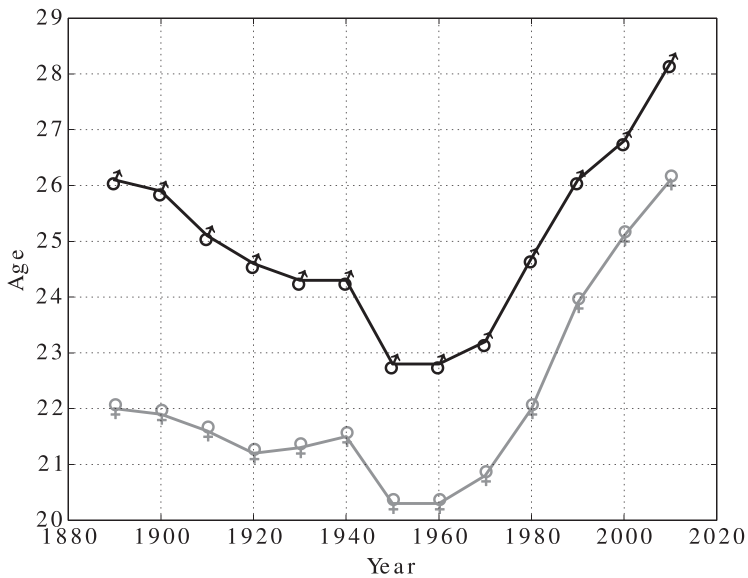


Figure 7.3 Median age at first marriage in the US, 1890–2010.

```
# Plot ages with male or female symbols as markers
ax.plot(year, age_m, marker='$\u2642$', markersize=14, c='blue', lw=2,
        mfc='blue', mec='blue')
ax.plot(year, age_f, marker='$\u2640$', markersize=14, c='magenta', lw=2,
        mfc='magenta', mec='magenta')
ax.grid()

ax.set_xlabel('Year')
ax.set_ylabel('Age')
ax.set_title('Median age at first marriage in the US, 1890 - 2010')

plt.show()
```

Example E7.3 The historical populations of five US cities are given in the files `boston.tsv`, `houston.tsv`, `detroit.tsv`, `san_jose.tsv`, `phoenix.tsv` as tab-separated columns of (year, population). They can be downloaded from scipython.com/eg/aaf.

The following program plots these data on one set of axes with a different line style for each.

Listing 7.3 The populations of five US cities over time

```
# eg7-populations.py
import matplotlib.pyplot as plt
import numpy as np

fig = plt.figure()
ax = fig.add_subplot(111)
```

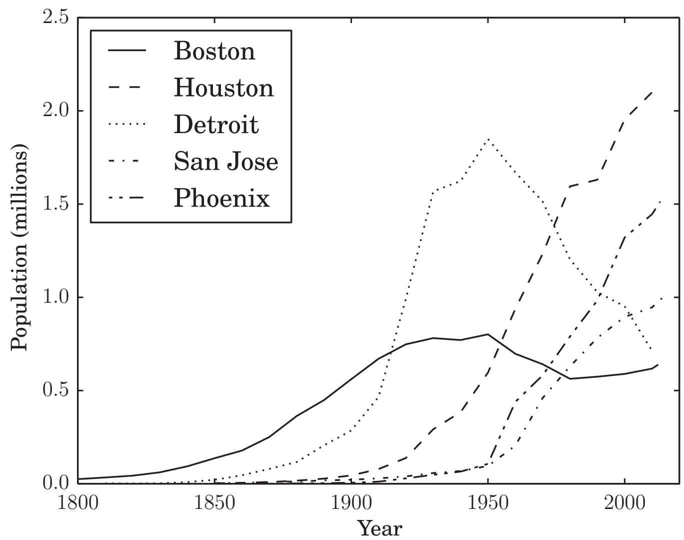


Figure 7.4 Population trends for five US cities.

```
cities = ['Boston', 'Houston', 'Detroit', 'San Jose', 'Phoenix']
# line styles: solid, dashes, dots, dash-dots, and dot-dot-dash
linestyles = [{'ls': '-'}, {'ls': '--'}, {'ls': ':'}, {'ls': '-.'},
              {'dashes': [2, 4, 2, 4, 8, 4]}]

for i, city in enumerate(cities):
❶   filename = '{}.tsv'.format(city.lower()).replace(' ', '_')
    yr, pop = np.loadtxt(filename, unpack=True)
    line, = ax.plot(yr, pop/1.e6, label=city, c='k', **linestyles[i])
ax.legend(loc='upper left')
ax.set_xlim(1800, 2020)
ax.set_xlabel('Year')
ax.set_ylabel('Population (millions)')
plt.show()
```

❶ Note how the city name is used to deduce the corresponding filename.

The plot produced is shown in Figure 7.4.

Font properties

The text elements of a plot (titles, legend, axis labels, etc.) can be customized with the arguments given in Table 7.6. For example,

```
ax.title('Plot Title', fontsize=18, fontname='Times New Roman', color='blue')
```

To use the same font properties for all text elements, it is easiest to set Matplotlib's `rc` settings using a dictionary of values. This involves a separate import from `pyplot` first:[3]

[3] It is also possible to edit Matplotlib's configuration file, `matplotlibrc`, to set many kinds of plot preferences: see http://matplotlib.org/users/customizing.html.

Table 7.6 Font property arguments for text elements of a plot

Argument	Description
`fontsize`	The size of the font in points (e.g., `12`, `16`)
`fontname`	The font name (e.g., `'Courier'`, `'Arial'`)
`family`	The font family (e.g., `'sans-serif'`, `'cursive'`, `'monospace'`)
`fontweight`	The font weight (e.g., `'normal'`, `'bold'`)
`fontstyle`	The font style (e.g., `'normal'`, `'italic'`)
`color`	Any Matplotlib color specifier (e.g., `'r'`, `'#ff00ff'`)

```
from matplotlib import rc
font_properties = {'family' : 'monospace',
                   'weight' : 'bold',
                   'size'   : 22}
rc('font', **font_properties)    ❶
# All text will now be rendered in 22-point, bold monospace in plots
```

❶ Recall that the syntax `**kwargs` passes the (key, value) pairs of dictionary `kwargs` and passes them to a function as keyword arguments (see Section 4.2.2).

Tick marks

Matplotlib does its best to label representative values (*tick marks*) on each axis appropriately, but there are some occasions when you want to customize them, for example, to make the tick marks more or less frequent, or to label them differently.

Most commonly, one simply wants to set the tick mark values to a given sequence of values: this is accomplished by calling `ax.set_xticks` and `ax.set_yticks` on the `Axes` object of the plot. For example,

```
ax.set_xticks([0, 1, 3.5, 6.5, 15])
```

Note that the ticks do not have to be evenly spaced.

To replace the actual numbered labels, pass a sequence of strings of a suitable length to `ax.set_xticklabels` and `ax.set_yticklabels`, as in the following example.[4]

Example E7.4 The following program plots the exponential decay described by $y = Ne^{-t/\tau}$ labeled by lifetimes, ($n\tau$ for $n = 0, 1, \cdots$) such that after each lifetime the value of y falls by a factor of e. The plot is given as Figure 7.5.

Listing 7.4 Exponential decay illustrated in terms of lifetimes

```
# eg7-ticks-exp-decay.py
import numpy as np
import matplotlib.pyplot as plt

# Initial value of y at t=0, lifetime in s
N, tau = 10000, 28
```

[4] Note that setting the tick labels directly in this way decouples your plot from its data to some extent. An entire module, `matplotlib.ticker`, is devoted to the configuration of tick locating and formatting: its API is beyond the scope of this book but is well described at http://matplotlib.org/api/ticker_api.html.

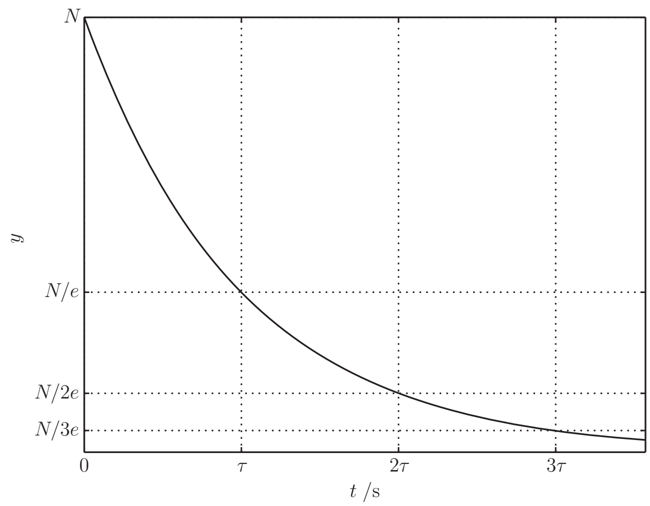


Figure 7.5 An exponential decay with customized tick labels.

```
# Maximum time to consider (s)
tmax = 100
# A suitable grid of time points, and the exponential decay itself
t = np.linspace(0, tmax, 1000)
y = N * np.exp(-t/tau)

fig = plt.figure()
ax = fig.add_subplot(111)
ax.plot(t, y)

# The number of lifetimes that fall within the plotted time interval
ntau = tmax // tau + 1
# xticks at 0, tau, 2*tau, ..., ntau*tau; yticks at the corresponding y-values
xticks = [i*tau for i in range(ntau)]
yticks = [N * np.exp(-i) for i in range(ntau)]
ax.set_xticks(xticks)
ax.set_yticks(yticks)

# xtick labels: 0, tau, 2tau, ...
xtick_labels = [r'$0$', r'$\tau$'] + [r'${}\tau$'.format(k) for k in range(2,ntau)]  ❶
ax.set_xticklabels(xtick_labels)
# corresponding ytick labels: N, N/e, N/2e, ...
ytick_labels = [r'$N$',r'$N/e$'] + [r'$N/{}e$'.format(k) for k in range(2,ntau)]  ❷
ax.set_yticklabels(ytick_labels)

ax.set_xlabel(r'$t\;/\mathrm{s}$')
ax.set_ylabel(r'$y$')
ax.grid()
plt.show()
```

❶ The x-axis tick labels are $0, \tau, 2\tau, \ldots$

❷ The y-axis tick labels are $N, N/e, N/2e, \ldots$

Note that the length of the sequence of tick labels must correspond to that of the list of tick values required.

Table 7.7 Common arguments to `ax.tick_params`

Argument	Description
`axis`	Which axis to customize: `'x'`, `'y'`, or `'both'`. Default is `'both'`.
`which`	Which tick mark set to customize: `'major'`, `'minor'`, or `'both'`. Default is `'major'`.
`direction`	Tick mark direction: `'in'`, `'out'`, or `'inout'`. Default is `'in'`.
`length`	Length of the tick marks in points.
`width`	Width of the tick marks in points.
`pad`	Distance between the tick mark and its label in points.
`labelsize`	Tick label size in points.
`color`	Tick mark color (a Matplotlib specifier).
`labelcolor`	Tick mark label color (a Matplotlib specifier).

To remove the tick labels altogether set them to the empty list, for example

```
ax.set_yticklabels([])
```

This retains the tick marks themselves. If you want neither tick marks nor tick labels on the axis use:

```
ax.set_yticks([])
```

There are two kinds of ticks: major ticks and minor ticks. Only major ticks are turned on by default; the smaller and more frequent minor ticks can most easily be enabled with

```
ax.minorticks_on()
```

More advanced customization of tick marks and their labels, including showing minor tick marks for one axis only, can be achieved using the `ax.tick_params` convenience function, which takes the arguments described in Table 7.7.

Finally, `ax.xaxis` and `ax.yaxis` have a method, `set_ticks_position`, which takes a single argument used to determine where the ticks appear: for `ax.xaxis`, `'top'`, `'bottom'`, `'both'` (the default) or `'none'`; for `ax.yaxis`, `'left'`, `'right'`, `'both'` (the default) or `'none'`.

Example E7.5 The following program creates a plot with both major and minor tick marks, customized to be thicker and wider than the default, where the major tick marks point into and out of the plot area.

Listing 7.5 Customized tick marks

```
# eg7-tick-customization.py

import numpy as np
import matplotlib.pyplot as plt

# A selection of functions on rn abcissa points for 0 <= x < 1
rn = 100
rx = np.linspace(0, 1, rn, endpoint=False)
```

```
def tophat(rx):
    """ Top hat function: y = 1 for x < 0.5, y=0 for x >= 0.5 """
    ry = np.ones(rn)
    ry[rx>=0.5]=0
    return ry

# A dictionary of functions to choose from
ry = {'half-sawtooth': lambda rx: rx.copy(),
      'top-hat': tophat,
      'sawtooth': lambda rx: 2 * np.abs(rx-0.5)}

# Repeat the chosen function nrep times
nrep = 4
x = np.linspace(0, nrep, nrep*rn, endpoint=False)
y = np.tile(ry['top-hat'](rx), nrep)    ❶

fig = plt.figure()
ax = fig.add_subplot(111)
ax.plot(x,y, 'k', lw=2)
# Add a bit of padding around the plotted line to aid visualization
ax.set_ylim(-0.1,1.1)
ax.set_xlim(x[0]-0.5, x[-1]+0.5)
# Customize the tick marks and turn the grid on
ax.minorticks_on()
ax.tick_params(which='major', length=10, width=2, direction='inout')
ax.tick_params(which='minor', length=5, width=2, direction='in')
ax.grid(which='both')
plt.show()
```

❶ This `np.tile` method constructs an array by repeating a given array `nrep` times. To plot a different periodic function, choose `'half-sawtooth'` or `'sawtooth'` here.

The resulting plot is shown in Figure 7.6.

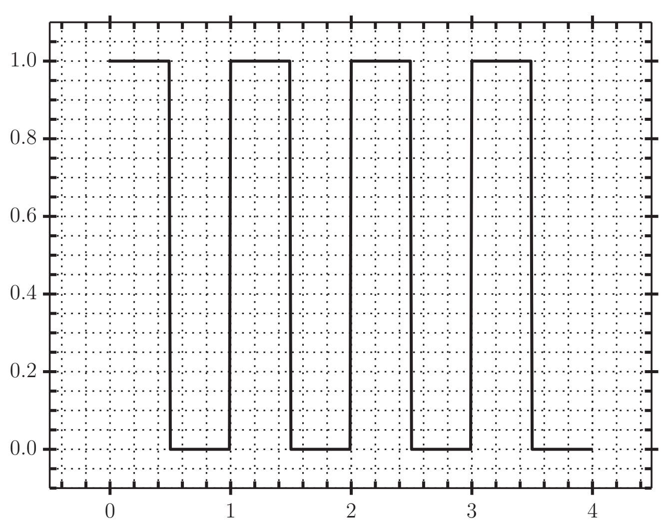


Figure 7.6 A periodic function plotted on a graph with gridlines and customized tick marks.

Table 7.8 Common arguments to `ax.errorbar`

Argument	Description
`x, y`	The data to plot
`yerr, xerr`	Errors on the x and y data coordinates, as described in the text
`fmt`	The plot format symbol (marker for the data point); set to `None` or the empty string, `''` to display only the error bars
`ecolor`	A Matplotlib color specifier for the error bars; the default, `None`, uses the same color as the connecting line between data markers
`elinewidth`	The width of the error bar lines in points; use `None` to use the same linewidth as the plotted data
`capsize`	The length of the error bar caps, in points
`errorevery`	A positive integer giving the subsampling for the error bars; for example, `errorevery=10` draws error bars on every 10th data point only

Error bars

To produce a plotted line with error bars, use the method `plt.errorbars` instead of `plt.plot`. In addition to the usual arguments of the `plot` function, `errorbars` allows the specification of errors in the $x-$ and $y-$ coordinates by passing the following types of value to the arguments `xerr` and `yerr`:

- `None`: No error bars for this coordinate;
- A scalar (e.g., `xerr=0.2`): all values are associated with symmetric error bars at plus and minus this value (i.e., ± 0.2);
- An array-type object of length `n` or shape `(n, 1)` (e.g., `yerr=[0.1, 0.15, 0.1]`): the symmetric error bars are plotted at plus and minus the values in this sequence for each of the `n` data points (i.e., $\pm 0.1, \pm 0.15, \pm 0.1$);
- An array-type object of shape `(2, n)` (i.e., two rows for each of `n` data points): error bars, which may be asymmetric, are plotted using minus-values from the first row and plus-values from the second.

The appearance of the error bars may be customized using the arguments summarized in Table 7.8. For example,

```
# Some data
x = array([ 0.3,  0.5,  0.7,  0.9])
y = array([ 1. ,  2. ,  2.5,  3.9])
# Constant, symmetric errors of +/- 0.05 on x-data
xerr = 0.05
# Asymmetric, variable errors on y-data
yerr = array([[ 0.1 ,  0.25,  0.5 ,  0.4 ],
              [ 0.1 ,  0.15,  0.2 ,  0.  ]])
ax.errorbar(x, y, yerr, xerr, fmt='o', ls='')
```

Example E7.6 Before fledging, some species of birds lose weight relative to the surface area of their wings to maximize their aerodynamic efficiency. The file

fledging-data.csv, available at scipython.com/eg/aad gives wing-loading values (body mass per wing area) as averages for two broods of swifts in the two weeks prior to fledging, with their uncertainties.[5]

In the program below, we perform a weighted fit to the data and plot it, with error bars.

Listing 7.6 Wing loading variation in swifts prior to fledging

```
# eg7-fledging.py
import numpy as np
import matplotlib.pyplot as plt

# Read in the data: day before fledging, wing loading and error for two broods
dt = np.dtype([('day', 'i2'), ('wl1', 'f8'), ('wl1-err', 'f8'),
               ('wl2', 'f8'), ('wl2-err', 'f8')])
data = np.loadtxt('fledging-data.csv', dtype=dt, delimiter=',')

# Weighted fit of exponential decay to the data. This is a linear least squares
# problem because y = Aexp(-Bx) => ln y = ln A - Bx = mx + c
p1_fit = np.poly1d(np.polyfit(data['day'], np.log(data['wl1']), 1,       ❶
                   w=np.log(data['wl1'])**-2))
p2_fit = np.poly1d(np.polyfit(data['day'], np.log(data['wl2']), 1,
                   w=np.log(data['wl2'])**-2))
wl1fit = np.exp(p1_fit(data['day']))
wl2fit = np.exp(p2_fit(data['day']))

# Plot the data points with their uncertainties and the fits
fig = plt.figure()
ax = fig.add_subplot(111)

# wl1 data: white circles, black borders, with error bars
ax.errorbar(data['day'], data['wl1'], yerr=data['wl1-err'], ls='', marker='o',
             color='k', mfc='w', mec='k')
ax.plot(data['day'], wl1fit, 'k', lw=1.5)

# wl2 data: black filled circles, with error bars
ax.errorbar(data['day'], data['wl2'], yerr=data['wl2-err'], ls='', marker='o',
             color='k', mfc='k', mec='k')
ax.plot(data['day'], wl2fit, 'k', lw=1.5)

ax.set_xlim(15,0)
ax.set_ylim(0.003, 0.012)
ax.set_xlabel('days pre-fledging')
ax.set_ylabel('wing loading ($\mathrm{g\,mm^{-2}}$)')
plt.show()
```

❶ The data points are weighted in the fit by $1/\sigma^2$ where σ is the estimated one-standard deviation error of the measurement.

Figure 7.7 shows the results of the fit. The broods, initially with different average wing-loading values, are seen to converge prior to fledging.

[5] J. Wright et al., *Proc. R. Soc. B* **273**, 1895 (2006).

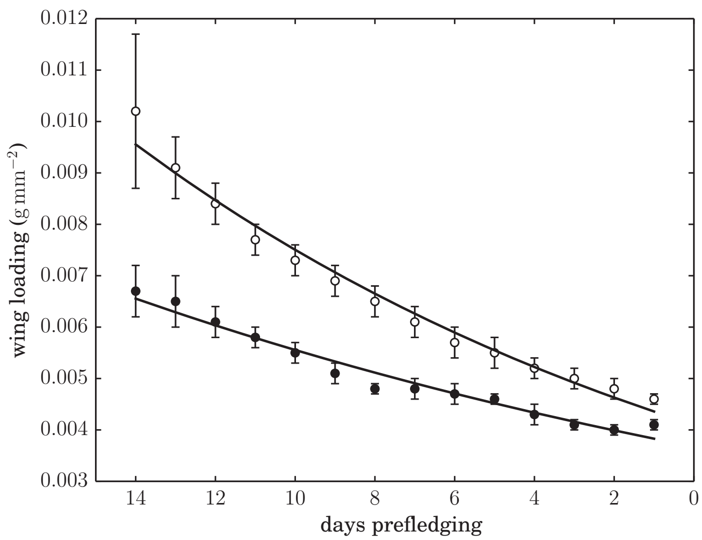


Figure 7.7 Fitted time series for wing-loading values in two cohorts of swift nestlings.

7.1.2 Bar charts and pie charts

Bar charts and histograms

The basic `pyplot` function for plotting a bar chart is `ax.bar`, which makes a plot of rectangular bars defined by their *left* edges and height. For example,

```
ax.bar([0, 1, 2], [40, 80, 20])
```

The width of the rectangles is, by default, `0.8` but can be set with the (third) `width` argument. If you want the bars vertically centered, either set the argument `align` to `'center'` or calculate where their left edges should be:

```
w = 0.5
x, y = np.array([0, 1, 2]), np.array([40, 80, 20])
ax.bar(x, y, w, align='center') # easiest way of centering the bars
ax.bar(x - w/2, y, w)           # or calculate the left edges
```

Additional arguments, including the provision of error bars, are given in Table 7.9.

By default, `ax.bar` produces a vertical bar chart. Horizontal bar charts are catered for either by setting `orientation='horizontal'` or by using the analogous `ax.barh` method.

Example E7.7 The following program produces a bar chart of letter frequencies in the English language, estimated by analysis of the text of *Moby-Dick*.[6] The vertical bars are centered and labeled by letter (Figure 7.8).

Listing 7.7 Letter frequencies in the text of *Moby-Dick*.

```
# eg7-charfreq.py
import numpy as np
import matplotlib.pyplot as plt
```

[6] See, for example, www.gutenberg.org/ebooks/2701 for a free text file of this novel.

Table 7.9 Common arguments to `ax.bar` and `barh`

Argument	Description
`left`	A sequence of x-coordinates of the left edges of the bars (but see `align`)
`height`	A sequence of heights for the bars
`width`	Width of the bars. If a scalar, all bars have the same width; can be array-like for variable widths
`bottom`	The y-coordinates of the bottom of the bars
`height`	A sequence of heights for the bars
`color`	Colors of the bar faces (scalar or array-like)
`edgecolor`	Colors of the bar edges (scalar or array-like)
`linewidth`	Line widths of the bar edges, in points (scalar or array-like)
`xerr, yerr`	Error bar limits, as for `errorbar` (scalar or array-like)
`error_kw`	A dictionary of keyword arguments corresponding to customization the appearance of the errorbars (see Table 7.8)
`align`	The default, `'edge'`, aligns the bars by their left edges (for vertical bars) or bottom edges (for horizontal bars); `'center'` centers the bars on this axis instead
`log`	Set to `True` to use a logarithmic axis scale
`orientation`	`'vertical'` (the default) or `'horizontal'`
`hatch`	Set the hatching pattern for the bars: one of `'/'`, `'\'`, `'|'`, `'-'`, `'+'`, `'x'`, `'o'`, `'O'`, `'.'`, `'*'`. Repeat the character for a denser pattern

```
text_file = 'moby-dick.txt'

letters = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
# Initialize the dictionary of letter counts: {'A': 0, 'B': 0, ...}
lcount = dict([(l, 0) for l in letters])

# Read in the text and count the letter occurrences
for l in open(text_file).read():
    try:
        lcount[l.upper()] += 1
    except KeyError:
        # Ignore characters that are not letters
        pass
# The total number of letters
norm = sum(lcount.values())

fig = plt.figure()
ax = fig.add_subplot(111)
# The bar chart, with letters along the horizontal axis and the calculated
# letter frequencies as percentages as the bar height
x = range(26)
ax.bar(x, [lcount[l]/norm * 100 for l in letters], width=0.8,
       color='g', alpha=0.5, align='center')
ax.set_xticks(x)
ax.set_xticklabels(letters)
ax.tick_params(axis='x', direction='out')
ax.set_xlim(-0.5, 25.5)
```

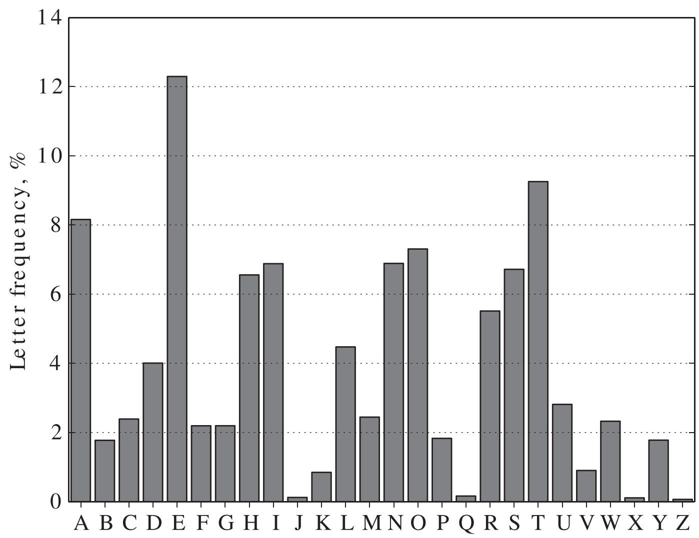


Figure 7.8 Letter frequencies in the novel *Moby-Dick*.

```
ax.yaxis.grid(True)
ax.set_ylabel('Letter frequency, %')
plt.show()
```

For monochrome plots, it is sometimes preferable to distinguish bars by patterns. The `hatch` argument can be used to do this, using any of several predefined patterns (see Table 7.9) as illustrated in the example below.

Example E7.8 The file `germany-energy-sources.txt`, available at scipython.com/eg/aae contains data on the renewable sources of electricity produced in Germany from 1990 to 2013:

```
Renewable electricity generation in Germany in GWh (million kWh)
Year      Hydro     Wind      Biomass   Photovoltaics
2013      21200     49800      47800     29300
2012      21793     50670      43350     26380
2011      17671     48883      37603     19559
...
```

The program below plots these data as a *stacked bar chart*, using Matplotlib's hatch patterns to distinguish between the different sources (Figure 7.9).

Listing 7.8 Visualizing renewable electricity generation in Germany

```
# eg7-germany-alt-energy.py
import numpy as np
import matplotlib.pyplot as plt

data = np.loadtxt('germany-energy-sources.txt', skiprows=2, dtype='i4')
years = data[:,0]
n = len(years)
```

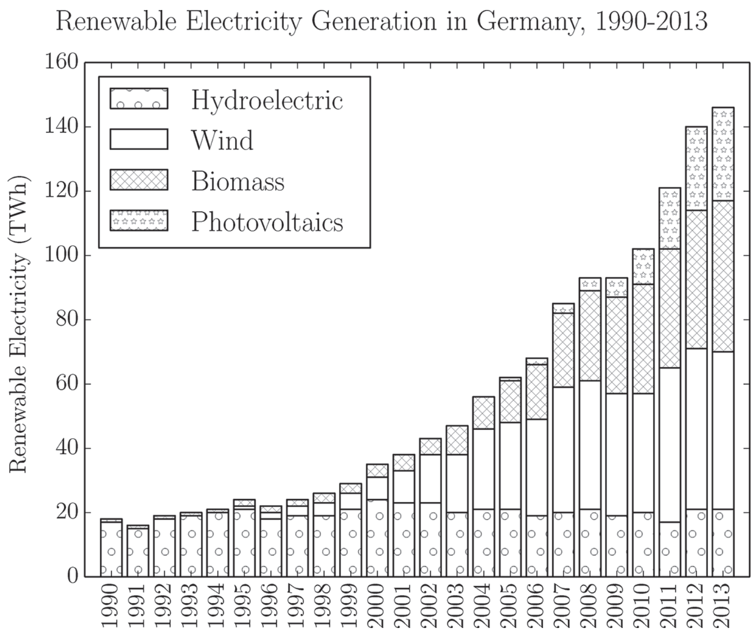


Figure 7.9 Stacked bar chart of renewable energy generation in Germany, 1990–2013.

```
# GWh to TWh
data[:,1:] /= 1000

fig = plt.figure()
ax = fig.add_subplot(111)
sources = ('Hydroelectric', 'Wind', 'Biomass', 'Photovoltaics')
hatch = ['o', '', 'xxxx', '**']
bottom = np.zeros(n)
bars = [None]*n
for i, source in enumerate(sources):
❶    bars[i] = ax.bar(years, bottom=bottom, height=data[:,i+1], color='w',
                     hatch=hatch[i], align='center')
    bottom += data[:,i+1]

ax.set_xticks(years)
plt.xticks(rotation=90)
ax.set_xlim(1989, 2014)
ax.set_ylabel('Renewable Electricity (TWh)')
ax.set_title('Renewable Electricity Generation in Germany, 1990-2013')
❷ plt.legend(bars, sources, loc='best')
plt.show()
```

❶ To include a legend, each bar chart object[7] must be stored in a list, `bars`, which ❷ is passed to the `ax.legend` method with a corresponding sequence of labels, `sources`.

[7] Actually a `Container` of *artists*.

Table 7.10 Common arguments to `ax.pie`

Argument	Description
`colors`	A sequence of Matplotlib color specifiers for coloring the segments
`labels`	A sequence of strings for labeling the segments
`explode`	A sequence of values specifying the fraction of the pie chart radius to offset each wedge by (0 for no explode effect)
`shadow`	`True` or `False`: specifies whether to draw an attractive shadow under the pie
`startangle`	Rotate the "start" of the pie chart by this number of degrees counter-clockwise from the horizontal axis
`autopct`	A format string to label the segments by their percentage fractional value, or a function for generating such a string from the data
`pctdistance`	The radial position of the `autopct` text, relative to the pie radius. The default is 0.6 (i.e., within the pie, which can be awkward for narrow segments)
`labeldistance`	The radial position of the `label` text, relative to the pie radius; the default is 1.1 (just outside the pie)
`radius`	The radius of the pie (the default is 1); this is useful when creating overlapping pie charts with different radii

Pie charts

It is straightforward to draw a pie chart in Matplotlib by passing an array of values to `ax.pie`. The values will be normalized by their sum if this sum is greater than 1, or otherwise treated directly as fractions. Labels, percentages, "exploded" segments and other effects are handled as described in Table 7.10 and illustrated in the following example.

Example E7.9 The following program depicts the emissions of greenhouse gases by mass of "carbon equivalent" (data from the 2007 IPCC report).[8]

Listing 7.9 Pie chart of greenhouse gas emissions

```
# eg7-pie.py
import numpy as np
import matplotlib.pyplot as plt

# Annual greenhouse gas emissions, billion tons carbon equivalent (GtCe)
gas_emissions = np.array([(r'$\mathrm{CO_2}$-d', 2.2),
                          (r'$\mathrm{CO_2}$-f', 8.0),
                          ('Nitrous\nOxide', 1.0),
                          ('Methane', 2.3),
                          ('Halocarbons', 0.1)],
                         dtype=[('source', 'U17'), ('emission', 'f4')])
```

[8] IPCC (2007), *Climate Change 2007: Synthesis Report. Contribution of Working Groups I, II and III to the Fourth Assessment Report of the Intergovernmental Panel on Climate Change* [Core Writing Team, Pachauri, R. K and Reisinger, A. (eds.)]. Geneva, Switzerland: IPCC, 104 pp.

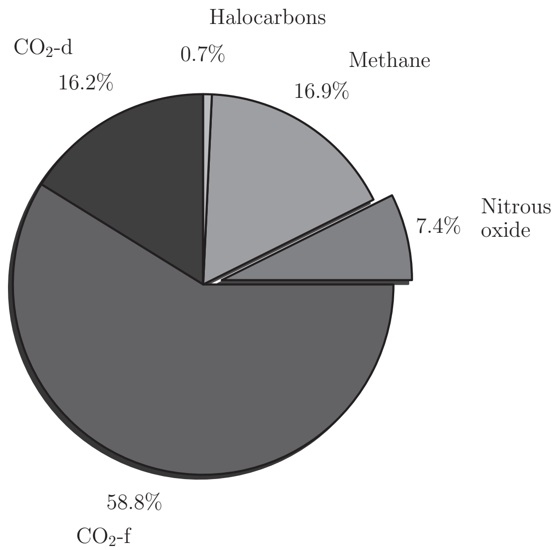


Figure 7.10 Greenhouse gas emissions by percentage for five different sources. CO_2-d denotes CO_2 emissions from deforestation; CO_2-f denotes CO_2 emissions from fossil fuel burning.

```
# 5 colors beige
colors = ['#C7B299', '#A67C52', '#C69C6E', '#754C24', '#534741']

explode = [0, 0, 0.1, 0, 0]                                          ❶

fig, ax = plt.subplots()
ax.axis('equal')        # So our pie looks round!
ax.pie(gas_emissions['emission'], colors=colors, shadow=True, startangle=90,
       explode=explode, labels=gas_emissions['source'], autopct='%.1f%%',   ❷
       pctdistance=1.15, labeldistance=1.3)

plt.show()
```

❶ The segment corresponding to nitrous oxide has been "exploded" by 10%.
❷ The percentage values are formatted to one decimal place (`autopct='%.1f%%'`). The resulting pie chart is shown in Figure 7.10.

7.1.3 Multiple subplots

To create a figure with more than one subplot (that is, `Axes`), call `add_subplot` on your `Figure` object, setting its argument to indicate where the subplot should be placed. Each call returns an `Axes` object. Single figures with more than 10 subplots are uncommon, so the usual argument is a three-digit number where each digit indicates the number of rows, number of columns and subplot number. The subplot number increases along the columns in each row and then down the rows. For example, a figure consisting of three rows of two columns of subplots can be constructed by adding `Axes` objects:

```
In [x]: fig = plt.figure()
In [x]: ax1 = fig.add_subplot(321)  # top left subplot
```

```
In [x]: ax2 = fig.add_subplot(322)  # top right subplot
In [x]: ax3 = fig.add_subplot(323)  # middle left subplot
...
In [x]: ax6 = fig.add_subplot(326)  # bottom right subplot
```

Alternatively, to create a figure and add all its subplots to it at the same time, call `plt.subplots`, which takes arguments `nrows` and `ncols` (in addition to those listed in Table 7.1) and returns a `Figure` and an array of `Axes` objects, which can be indexed for each individual axis:

```
In [x]: fig, axes = plt.subplots(nrows=3, ncols=2)
In [x]: axes.shape
Out[x]: (3, 2)

In [x]: ax1 = axes[0,0]       # top left subplot
In [x]: ax2 = axes[2,1]       # bottom right subplot
```

In fact, a useful idiom to create a plot with a single `Axes` object is to call `subplots()` with no arguments:

```
In [x]: fig, ax = plt.subplots()
In [x]: ax.plot(x,y)        # no need to index the single Axes object created
```

Plots with subplots run the risk of their labels, titles and ticks overlapping each other – if this happens, call the method `tight_layout` on the `Figure` object and Matplotlib will do its best to arrange them so that there is sufficient space between them.

Example E7.10 Consider a metal bar of cross-sectional area, A, initially at a uniform temperature, θ_0, which is heated instantaneously at the exact center by the addition of an amount of energy, H. The subsequent temperature of the bar (relative to θ_0) as a function of time, t, and position, x, is governed by the one-dimensional diffusion equation:

$$\theta(x,t) = \frac{H}{c_pA}\frac{1}{\sqrt{Dt}}\frac{1}{\sqrt{4\pi}}\exp\left(-\frac{x^2}{4Dt}\right),$$

where c_p and D are the metal's specific heat capacity per unit volume and thermal diffusivity (which we assume are constant with temperature). The following code plots $\theta(x,t)$ for three specific times and compares the plots between two metals, with different thermal diffusivities but similar heat capacities, copper and iron.

Listing 7.10 The one-dimensional diffusion equation applied to the temperature of two different metal bars

```
# eg7-diffusion1d.py
import numpy as np
import matplotlib.pyplot as plt

# Cross-sectional area of bar in m3, heat added at x=0 in J
A, H = 1.e-4, 1.e3
# Temperature in K at t=0
theta0 = 300

# Metal element symbol, specific heat capacities per unit volume (J.m-3.K-1),
# Thermal diffusivities (m2.s-1) for Cu and Fe
```

```
metals = np.array([('Cu', 3.45e7, 1.11e-4), ('Fe', 3.50e7, 2.3e-5)],
                  dtype=[('symbol', '|S2'), ('cp', 'f8'), ('D', 'f8')])

# The metal bar extends from -xlim to xlim (m)
xlim, nx = 0.05, 1000
x = np.linspace(-xlim, xlim, nx)

# Calculate the temperature distribution at these three times
times = (1e-2, 0.1, 1)
# Create our subplots: three rows of times, one column for each metal
fig, axes = plt.subplots(nrows=3, ncols=2, figsize=(7, 8))
for j, t in enumerate(times):
    for i, metal in enumerate(metals):
        symbol, cp, D = metal
        ax = axes[j, i]
        # The solution to the diffusion equation
        theta = theta0 + H/cp/A/np.sqrt(D*t) / 4/np.pi * np.exp(-x**2/4/D/t)
        # Plot, converting distances to cm and add some labeling
        ax.plot(x*100, theta, 'k')
        ax.set_title('{}, $t={}$ s'.format(symbol.decode('utf8'), t))
        ax.set_xlim(-4, 4)
        ax.set_xlabel('$x\;/\mathrm{cm}$')
        ax.set_ylabel('$\Theta\;/\mathrm{K}$')

# Set up the y axis so that each metal has the same scale at the same t
for j in (0,1,2):
    ymax = max(axes[j,0].get_ylim()[1], axes[j,1].get_ylim()[1])
    print(axes[j,0].get_ylim(), axes[j,1].get_ylim())
    for i in (0,1):
        ax = axes[j,i]
        ax.set_ylim(theta0, ymax)
        # Ensure there are only three y-tick marks
        ax.set_yticks([theta0, (ymax + theta0)/2, ymax])
# We don't want the subplots to bash into each other: tight_layout() fixes this
fig.tight_layout()
plt.show()
```

Because copper is a better conductor, the temperature increase is seen to spread more rapidly for this metal (see Figure 7.11).

To further customize the subplot spacing, call `fig.subplots_adjust()`. This method takes any of the keywords `left`, `bottom`, `right`, `top`, `wspace` and `hspace`, which can be set to fractional values of the figure's height and width as appropriate to determine the positions of the subplots' left side (default 0.125), right side (0.9), bottom (0.1), top (0.9), vertical spacing (0.2) and horizontal spacing (0.2). A practical use of this function is to create "ganged" subplots that share a common axis, as in the following example.

Example E7.11 This code generates a figure of 10 subplots depicting the graph of $\sin(n\pi x)$ for $n = 0, 1, \cdots 9$. The subplot spacing is configured so that they "run into" each other vertically (see Figure 7.12).

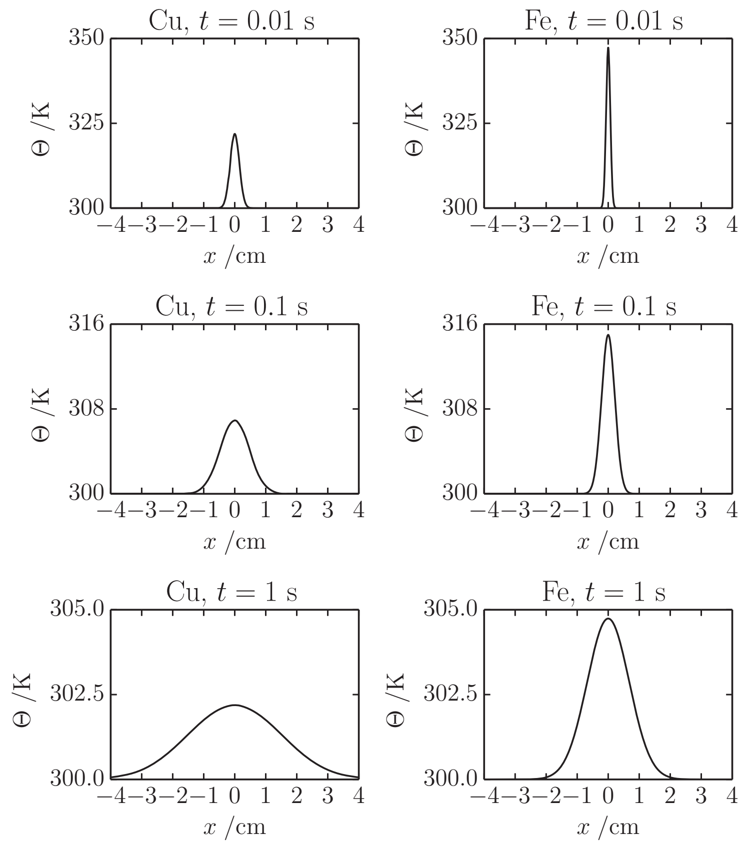


Figure 7.11 Numerical solutions of the one-dimensional diffusion equation for the temperatures of two metal bars.

Listing 7.11 Ten subplots with zero vertical spacing

```
import numpy as np
import matplotlib.pyplot as plt

nrows = 10
fig, axes = plt.subplots(nrows,1)
# Zero vertical space between subplots
fig.subplots_adjust(hspace=0)

x = np.linspace(0,1,1000)
for i in range(nrows):
    # n=nrows for the top subplot, n=0 for the bottom subplot
    n = nrows - i
    axes[i].plot(x, np.sin(n * np.pi * x), 'k', lw=2)
    # We only want ticks on the bottom of each subplot
    axes[i].xaxis.set_ticks_position('bottom')
    if i < nrows-1:
        # Set ticks at the nodes (zeros) of our sine functions
        axes[i].set_xticks(np.arange(0, 1, 1/n))
        # We only want labels on the bottom subplot xaxis
        axes[i].set_xticklabels('')
    axes[i].set_yticklabels('')
plt.show()
```

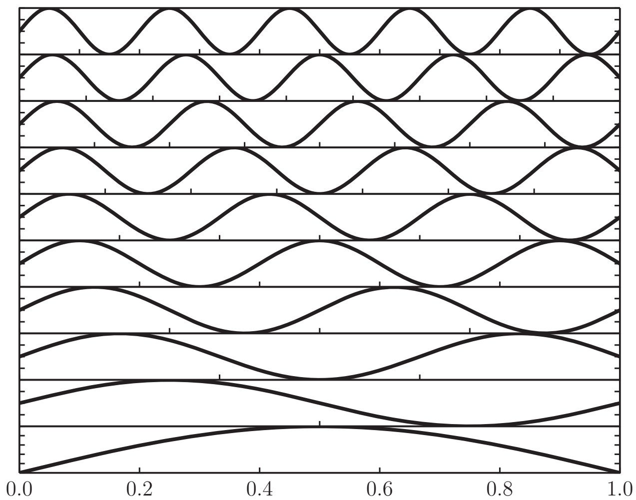


Figure 7.12 Ten subplots of $\sin(n\pi x)$ for $n = 0, 1, \cdots 9$ adjusted to remove vertical space between them.

7.1.4 Annotations

Matplotlib provides several ways to add different kinds of annotation to your plots. The most important methods for adding text, arrows, lines and shapes are described below.

Adding text

The method `ax.text(x,y,s)` is a basic method used to add a text string `s` at position `(x,y)` (in *data* coordinates) to the axes. The font properties can be determined by additionally passing a dictionary of (keyword, value) pairs to `fontdict` (see Table 7.6). Individual keyword arguments (such as `fontsize=20`) can also be used to customize the font in this way.

If the text annotation refers to a feature of the data, you will usually want the default behavior, placing it using data coordinates so that it maintains the same relative position to the data even if the plot limits are changed. If, instead, you want to place the text in *axis* coordinates (such that (0,0) is the lower left of the axes and (1,1) is the upper right), set the keyword argument `transform=ax.transAxes` where `ax` is the `Axes` object the coordinates refer to.

Arrows and text

The `ax.annotate` method is similar to `ax.text` (although with an annoyingly different syntax) but draws an arrow from the text to a specified point in the plot. The important arguments to `ax.annotate` are

- `s`, the string to output as a text label;
- `xy`, a tuple, `(x,y)` giving the coordinates of the position to annotate (i.e., where the arrow points *to*);
- `xytext`, a tuple, `(x,y)` giving the coordinates of the text label (i.e., where the arrow points *from*);

- `xycoords`, an optional string determining the type of coordinates referred to by the argument `xy`: several options are available,[9] but the most commonly used ones are
  - `'data'`: data coordinates, the default,
  - `'figure fraction'`: fractional coordinates of the *figure size* (`(0,0)` is lower left, `(1,1)` is upper right),
  - `'axes fraction'`: fractional coordinates of the *axes* (`(0,0)` is lower left, `(1,1)` is upper right), and
- `textcoords`: as for `xycoords`, an optional string determining the type of coordinates referred to by `xytext`. An additional value is permitted for this string: `'offset points'` specifies that the tuple `xytext` is an offset *in points* from the `xy` position.
- `arrowprops`: if present, determines the properties and style of the arrow drawn between `xytext` and `xy` (see below).

Additional keyword arguments are interpreted as properties of the `Text` object produced as the label (e.g., `fontsize` and `color`). An important pair is `verticalalignment` (or `va`) and `horizontalalignment` (or `ha`) which determine how the label is aligned relative to its `xytext` position. Valid values are `'center'`, `'right'`, `'left'`, `'top'`, `'bottom'` and `'baseline'` as appropriate.

In its simplest usage, `ax.annotate` just adds a text label to the plot (without an arrow). For example,

```
ax.annotate('My Label', xy=(0.5,0.8), fontsize=16, xycoords='axes fraction',
            ha='center')
```

which adds `'My Label'` at the center, near the top of the axes in 16-point text. Note that if there is no arrow or line, `xytext` is not necessary and the label is placed directly at `xy`.

The argument `arrowprops` is a dictionary determining the style of line or arrow joining the label at `xytext` to the specified `xy` point. There is a somewhat bewildering array of possible items to put in this dictionary, but the important ones are illustrated by the following example.

Example E7.12 The following program produces a plot with eight arrows with different styles (Figure 7.13).

Listing 7.12 Annotations with arrows in Matplotlib

```
# eg7-arrows.py
import numpy as np
import matplotlib.pyplot as plt

fig, ax = plt.subplots()
x = np.linspace(0,1)
ax.plot(x, x, 'o')
```

[9] See the documentation at http://matplotlib.org/api/text_api.html/matplotlib.text.Annotation.

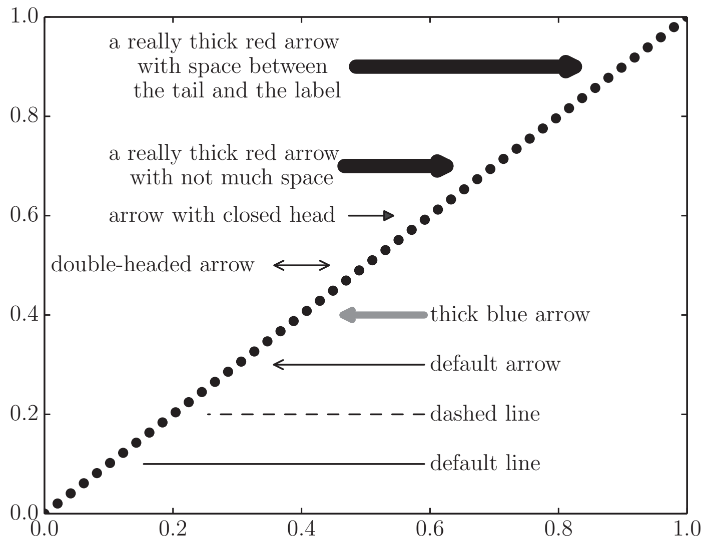


Figure 7.13 An example of different arrow styles.

```
ax.annotate('default line', xy=(0.15,0.1), xytext=(0.6,0.1),
            arrowprops={'arrowstyle': '-'}, va='center')
ax.annotate('dashed line', xy=(0.25,0.2), xytext=(0.6,0.2),
            arrowprops={'arrowstyle': '-', 'ls': 'dashed'}, va='center')
ax.annotate('default arrow', xy=(0.35,0.3), xytext=(0.6,0.3),
            arrowprops={'arrowstyle': '->'}, va='center')
ax.annotate('thick blue arrow', xy=(0.45,0.4), xytext=(0.6,0.4),
            arrowprops={'arrowstyle': '->', 'lw': 4, 'color': 'blue'},
            va='center')
ax.annotate('double-headed arrow', xy=(0.45,0.5), xytext=(0.01,0.5),
            arrowprops={'arrowstyle': '<->'}, va='center')
ax.annotate('arrow with closed head', xy=(0.55,0.6), xytext=(0.1,0.6),
            arrowprops={'arrowstyle': '-|>'}, va='center')
ax.annotate('a really thick red arrow\nwith not much space', xy=(0.65,0.7),
            xytext=(0.1,0.7), va='center', multialignment='right',
            arrowprops={'arrowstyle': '-|>', 'lw': 8, 'ec': 'r'})
ax.annotate('a really thick red arrow\nwith space between\nthe tail and the'
            'label', xy=(0.85,0.9), xytext=(0.1,0.9), va='center',
            multialignment='right',
            arrowprops={'arrowstyle': '-|>', 'lw': 8, 'ec': 'r', 'shrinkA': 10})

plt.show()
```

Example E7.13 Another example of an annotated plot, this time of the share price of BP plc (LSE: BP) with a couple of notable events added to it. The necessary data for this example can be downloaded from Yahoo! Finance.[10]

[10] https://uk.finance.yahoo.com/q/hp?s=BP.L.

Listing 7.13 eg7-share-prices

```
import datetime
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.dates import strpdate2num
from datetime import datetime

❶ def date_to_int(s):
    epoch = datetime(year=1970, month=1, day=1)
    date = datetime.strptime(s, '%Y-%m-%d')
    return (date - epoch).days

def bindate_to_int(bs):
    return date_to_int(bs.decode('ascii'))

dt = np.dtype([('daynum','i8'), ('close', 'f8')])
share_price = np.loadtxt('bp-share-prices.csv', skiprows=1, delimiter=',',
                      usecols=(0,4), converters={0: bindate_to_int},
                      dtype=dt)
fig = plt.figure()
ax = fig.add_subplot(111)
ax.plot(share_price['daynum'], share_price['close'], c='g')
❷ ax.fill_between(share_price['daynum'], 0, share_price['close'], facecolor='g',
                alpha=0.5)

daymin, daymax = share_price['daynum'].min(), share_price['daynum'].max()
ax.set_xlim(daymin, daymax)

price_max = share_price['close'].max()

def get_xy(date):
    """ Return the (x,y) coordinates of the share price on a given date. """
    x = date_to_int(date)
    return share_price[np.where(share_price['daynum']==x)][0]

# A horizontal arrow and label
x,y = get_xy('1999-10-01')
ax.annotate('Share split', (x,y), xytext = (x+1000,y), va='center',
            arrowprops=dict(facecolor='black', shrink=0.05))
# A vertical arrow and label
x,y = get_xy('2010-04-20')
ax.annotate('Deepwater Horizon\noil spill', (x,y), xytext = (x,price_max*0.9),
            arrowprops=dict(facecolor='black', shrink=0.05), ha='center')

years = range(1989,2015,2)
ax.set_xticks([date_to_int('{:4d}-01-01'.format(year)) for year in years])
❸ ax.set_xticklabels(years, rotation=90)

plt.show()
```

❶ We need to do some work to read in the date column: first decode the byte string read in from the file to ASCII (`bindate_to_int`), then use `datetime` (see Section 4.5.3) to convert it into an integer number of days since some reference date (epoch): here we choose the Unix epoch, 1 January 1970 (`date_to_int`).

❷ `ax.fill_between` fills the region below the plotted line with a single color.

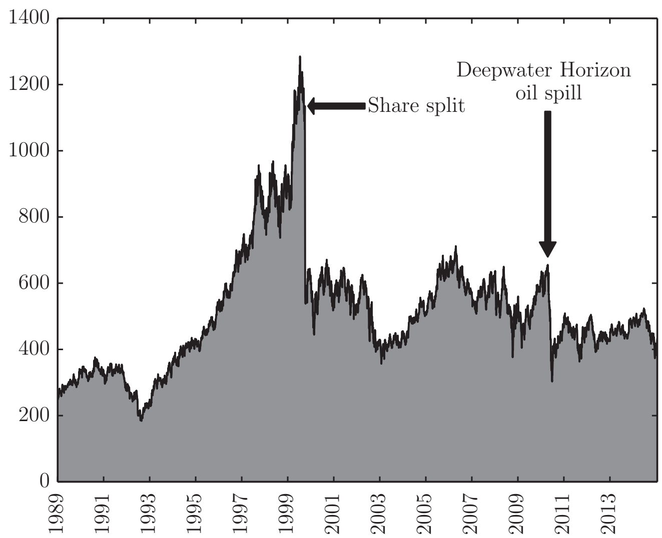


Figure 7.14 BP plc's share price on an annotated chart.

❸ We rotate the year labels so there's enough room for them (reading bottom to top). Figure 7.14 shows the plotted chart.

Lines and span rectangles

Adding an arbitrary straight line to a Matplotlib plot can be achieved by simply plotting the data corresponding to its start and end points with `ax.plot`; for example,

```
ax.plot([x1, x2], [y1, y2], color='k', lw=2)
```

draws a line between `(x1, y1)` and `(x2, y2)`. Of course, this approach would be tedious for a drawing a large number of disconnected lines, so for horizontal and vertical lines there are a pair of convenient methods, `ax.hlines` and `ax.vlines`. `ax.hlines` takes mandatory arguments `y`, `xmin`, `xmax` and draws horizontal lines with *y*-coordinates at each of the values given by the sequence `y` (if `y` is passed as a scalar, a single line is drawn). `xmin` and `xmax` specify the start and end of each line; they can be scalars (in which case all the lines will have the same start and end *x*-coordinates) or a sequence (with one value for each of the *y*-coordinates specified by `y`). `ax.vlines` draws vertical lines; its mandatory arguments, `x`, `ymin` and `ymax` are entirely analogous.

Example E7.14 The code below illustrates some different uses of `ax.vlines` and `ax.hlines` (see Figure 7.15).

Listing 7.14 Some different ways to use `ax.vlines` and `ax.hlines`

```
# eg7-circle-lines.py
import numpy as np
import matplotlib.pyplot as plt

fig, ax = plt.subplots()
ax.axis('equal')
```

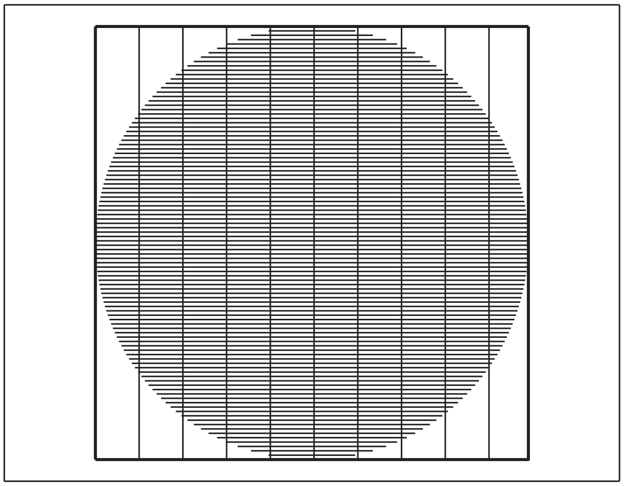


Figure 7.15 A figure generated from vertical and horizontal lines.

```
# A circle made of horizontal lines
y = np.linspace(-1,1,100)
xmax = np.sqrt(1 - y**2)
ax.hlines(y, -xmax, xmax, color='g')

# Draw a box of thicker lines around the circle
ax.vlines(-1, -1, 1, lw=2, color='r')
ax.vlines(1, -1, 1, lw=2, color='r')
ax.hlines(-1, -1, 1, lw=2, color='r')
ax.hlines(1, -1, 1, lw=2, color='r')
# Some evenly spaced vertical lines
ax.vlines(y[::10], -1, 1, color='b')

# Remove tick marks and labels
ax.xaxis.set_visible(False)
ax.yaxis.set_visible(False)
# A bit of padding around the outside of the box
ax.set_xlim(-1.1,1.1)
ax.set_ylim(-1.1,1.1)

plt.show()
```

On static plots such as figures for printing, `ax.hlines` and `ax.vlines` work well, but note that the line limits don't change upon changing the axes' limits in an interactive plot. There are two further methods, `ax.axhline` and `ax.axvline`, which simply plot a horizontal or vertical line across the axis, whatever its current limits. `axhline` takes arguments `y`, `xmin`, `xmax`, but these must be scalar values (so multiple lines require repeated calls) and `xmin`, `xmax` are given in *fractional* coordinates such that 0 is the far left of the plot and 1 the far right. Again, the `axvline` arguments: `x`, `ymin`, `ymax` are analogous. Some examples:

```
ax.axhline(100, 0, 1)    # Horizontal line across whole of x-axis at y = 100.
ax.axhline(100)          # Same thing: xmin and xmax default to 0 and 1
```

Table 7.11 Keyword arguments for styling patches

Argument	Description
`alpha`	Set the alpha transparency (0-1)
`color`	Set both the `facecolor` and the `edgecolor` of the patch
`edgecolor`, `ec`	Set the edge (border) color
`facecolor`, `fc`	Set the patch face color
`fill`	Indicate whether to fill the patch or not (`True` or `False`)
`hatch`	Set the hatching pattern for the patch: one of `'/'`, `'\'`, `'|'`, `'-'`, `'+'`, `'x'`, `'o'`, `'O'`, `'.'`, `'*'`. Repeat the character for a denser pattern
`linestyle`, `ls`	Set the patch line style: `'solid'`, `'dashed'`, `'dashdot'`, `'dotted'`
`linewidth`, `lw`	Set the patch line width, in points

```
# A thick, blue, dashed vertical line at x = 5. around the center of the y-axis
ax.axvline(5, 0.4, 0.6, c='b', lw=4, ls='--')
```

The methods `ax.axhspan` and `ax.axvspan` are similar but produce a horizontal or vertical *spanning rectangle* across the axis. `ax.axhspan` is passed arguments `ymin`, `ymax` (in *data* coordinates), and `xmin`, `xmax` (in *fractional axes* units). `ax.axvspan` takes the arguments `xmin`, `xmax`, `ymin` and `ymax` analogously. Extra keywords can be used to style the spanning rectangle (see Table 7.11).

Example E7.15 The program below annotates a simple wave plot to indicate the different regions of the electromagnetic spectrum, using `text`, `axvline`, `axhline` and `axvspan` (see Figure 7.16).

Listing 7.15 A representation of the electromagnetic spectrum, 250–1,000 nm

```
# eg7-annotate.py
import numpy as np
import matplotlib.pyplot as plt

# wavelength range, nm
lmin, lmax = 250, 1000
x = np.linspace(lmin, lmax, 1000)
# A wave with a smoothly increasing wavelength
wv = (np.sin(10 * np.pi * x / (lmax+lmin-x)))[::-1]

fig = plt.figure()
ax = fig.add_subplot(111, axisbg='k')
ax.plot(x, wv, c='w', lw=2)
ax.set_xlim(250,1000)
ax.set_ylim(-2,2)

# Label and delimit the different regions of the electromagnetic spectrum
ax.text(310, 1.5, 'UV', color='w', fontdict={'fontsize': 20})
ax.text(530, 1.5, 'Visible', color='k', fontdict={'fontsize': 20})
ax.annotate('', (400, 1.3), (750, 1.3), arrowprops={'arrowstyle': '<|-|>',
                                                    'color': 'w', 'lw': 2})
ax.text(860, 1.5, 'IR', color='w', fontdict={'fontsize': 20})
ax.axvline(400, -2, 2, c='w', ls='--')
```

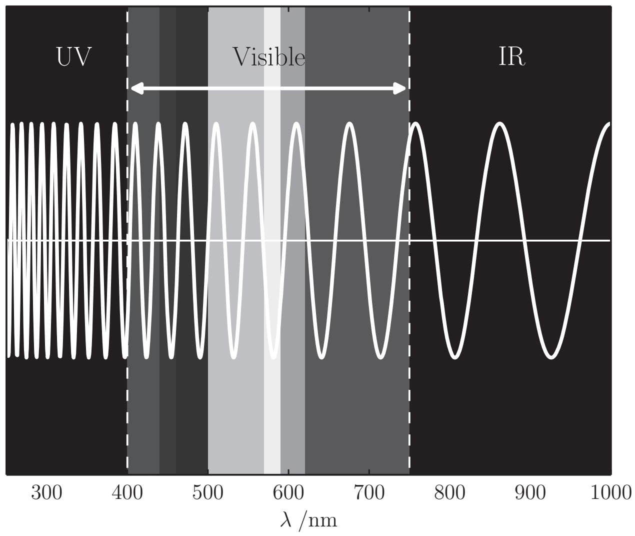


Figure 7.16 A representation of the electromagnetic spectrum.

```
ax.axvline(750, -2, 2, c='w', ls='--')
# Horizontal "axis" across the center of the wave
ax.axhline(c='w')
# Ditch the y-axis ticks and labels; label the x-axis
ax.yaxis.set_visible(False)
ax.set_xlabel(r'$\lambda\;/\mathrm{nm}$')

# Finally, add some colorful rectangles representing a rainbow in the
# visible region of the spectrum.
# Dictionary mapping of wavelength regions (nm) to approximate RGB values
rainbow_rgb = { (400, 440): '#8b00ff', (440, 460): '#4b0082',
                (460, 500): '#0000ff', (500, 570): '#00ff00',
                (570, 590): '#ffff00', (590, 620): '#ff7f00',
                (620, 750): '#ff0000'}
for wv_range, rgb in rainbow_rgb.items():
    ax.axvspan(*wv_range, color=rgb, ec='none', alpha=1)
plt.show()
```

7.1.5 ◇ Circles, ellipses, rectangles and other patches

Almost everything that gets rendered in a Matplotlib figure is a subclass of the abstract base class, `Artist`. This includes lines (through `Line2D`) and text (through `Text`).[11] An important collection of rendered objects is further derived from the `Artist` subclass `Patch`: a two-dimensional shape. The wedges of a pie chart (Section 7.1.2) and the arrows of an annotation (Section 7.1.4) are examples we have met before.

[11] In fact, there are two kinds of `Artist`: *primitives* and *containers*. Primitives are the graphical objects (such as `Line2D` themselves) and containers are the elements of a figure onto which they are rendered (for example `Axes`).

To add a shape to an `Axes` object, create a patch using one of the classes described in full in the Matplotlib documentation[12] and call `ax.add_artist(patch)`. To set the color, line widths, transparency, etc. of the patch, pass one or more of the keywords listed in Table 7.11 when creating the patch.

Below we describe this usage for a few `Patch` objects.

Circles and Ellipses

A `Circle` centered at `xy = (x,y)` (in data coordinates) and with radius `r` is created with:

```
from matplotlib.patches import Circle
circle = Circle(xy, r, **kwargs)
```

It is added to the Axes with `ax.add_artist`:

```
ax.add_artist(circle)
```

The supported keyword arguments indicated by `**kwargs` are the usual patch styling ones, summarized in Table 7.11.

`Ellipse` patches are similar but take arguments `width` and `height` (the total length of the horizontal and vertical axes of the ellipse before rotation) and `angle` (the angle of counterclockwise rotation of the ellipse *in degrees*).

```
from matplotlib.patches import Ellipse
ellipse = Ellipse(xy, width, height, angle, **kwargs)
```

Example E7.16 The following code reads in the heights and masses of 260 women and 247 men from the data set published by Heinz *et al.*[13] and available for download at www.amstat.org/publications/jse/datasets/body.dat.txt. It plots the (height, mass) pairs for each individual on a scatterplot and, for each sex, draws a 3σ covariance ellipse around the mean point. The dimensions of this ellipse are given by the (scaled) eigenvalues of the covariance matrix and it is rotated such that its semi-major axis lies along the largest eigenvector.

Listing 7.16 An analysis of the height-mass relationship in 507 healthy individuals

```
# eg7-body-mass-height.py
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.patches import Ellipse

FEMALE, MALE = 0, 1
dt = np.dtype([('mass', 'f8'), ('height', 'f8'), ('gender', 'i2')])
data = np.loadtxt('body.dat.txt', usecols=(22,23,24), dtype=dt)

fig, ax = plt.subplots()
```

---

[12] http://matplotlib.org/api/artist_api.html.

[13] G. Heinz *et al.*, *Journal of Statistical Education* **11** (2), 2003. This article is available at www.amstat.org/publications/jse/v11n2/datasets.heinz.html.

```
def get_cov_ellipse(cov, center, nstd, **kwargs):
    """
    Return a matplotlib Ellipse patch representing the covariance matrix
    cov centered at center and scaled by the factor nstd.

    """

    # Find and sort eigenvalues and eigenvectors into descending order
    eigvals, eigvecs = np.linalg.eigh(cov)
    order = eigvals.argsort()[::-1]
    eigvals, eigvecs = eigvals[order], eigvecs[:, order]

    # The counterclockwise angle to rotate our ellipse by
    vx, vy = eigvecs[:,0][0], eigvecs[:,0][1]
❶   theta = np.arctan2(vy, vx)

    # Width and height of ellipse to draw
    width, height = 2 * nstd * np.sqrt(eigvals)
    return Ellipse(xy=center, width=width, height=height,
                   angle=np.degrees(theta), **kwargs)

labels, colors =['Female', 'Male'], ['magenta', 'blue']
for gender in (FEMALE, MALE):
    sdata = data[data['gender']==gender]
    height_mean = np.mean(sdata['height'])
    mass_mean = np.mean(sdata['mass'])
    cov = np.cov(sdata['mass'], sdata['height'])
    ax.scatter(sdata['height'], sdata['mass'], color=colors[gender],
               label=labels[gender])
    e = get_cov_ellipse(cov, (height_mean, mass_mean), 3,
                        fc=colors[gender], alpha=0.4)
    ax.add_artist(e)

ax.set_xlim(140, 210)
ax.set_ylim(30, 120)
ax.set_xlabel('Height /cm')
ax.set_ylabel('Mass /kg')
ax.legend(loc='upper left', scatterpoints=1)
plt.show()
```

❶ The function `np.arctan2` returns the "two-argument arctangent": `np.arctan2(y, x)` is the angle in radians between the positive x-axis and the point (x, y).

Figure 7.17 shows the resulting plot.

Rectangles

`Rectangle` patches are created in a similar way to `Ellipses`:

```
from matplotlib.patches import Rectangle
rectangle = Rectangle(xy, width, height, angle, **kwargs)
```

Here, however, the tuple `xy=(x,y)` gives the coordinates of the *lower-left* corner of the rectangle. A square is simply a rectangle with the same `width` and `height`, of course.

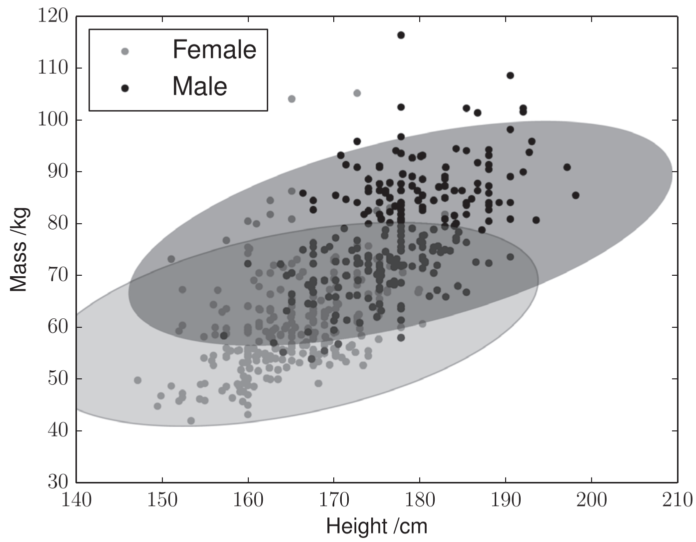


Figure 7.17 Scatterplots for each gender of mass and height for a total of 507 students, with their covariance ellipses annotated.

Polygons

A `Polygon` patch is created by passing an array of shape `(N,2)`, in which each row represents the (x, y) coordinates of a vertex. If the additional argument, `closed` is `True` (the default), the polygon will be closed so that the start and end points are the same. This is illustrated in the following example.

Example E7.17 This code produces an image (Figure 7.18) of some colorful shapes.

Listing 7.17 Some colorful shapes

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.patches import Polygon, Circle, Rectangle

red, blue, yellow, green = '#ff0000', '#0000ff', '#ffff00', '#00ff00'
square = Rectangle((0.7, 0.1), 0.25, 0.25, facecolor=red)
circle = Circle((0.8, 0.8), 0.15, facecolor=blue)
triangle = Polygon(((0.05,0.1), (0.396,0.1), (0.223, 0.38)), fc=yellow)
rhombus = Polygon(((0.5,0.2), (0.7,0.525), (0.5,0.85), (0.3,0.525)),  fc=green)

fig = plt.figure()
ax = fig.add_subplot(111, axisbg='k', aspect='equal')
for shape in (square, circle, triangle, rhombus):
    ax.add_artist(shape)
ax.xaxis.set_visible(False)
ax.yaxis.set_visible(False)

plt.show()
```

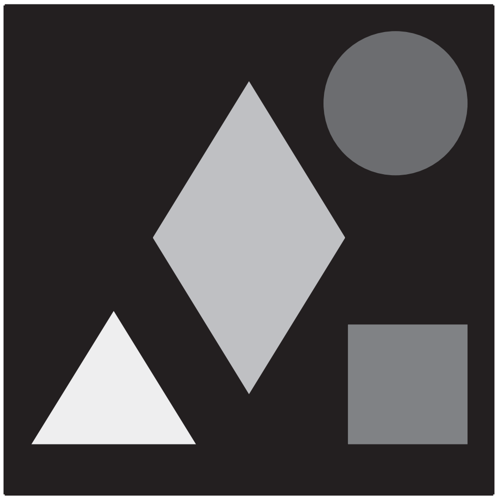


Figure 7.18 Some colorful shapes using Matplotlib patches.

Questions

Q7.1.1 Compare plots of $y = x^3$ for $-10 \leq x \leq 10$ using a logarithmic scale on the x-axis, y-axis and both axes. What is the difference between using `ax.set_xscale('log')` and `ax.set_xscale('symlog')`?

Q7.1.2 Adapt Example E7.7 to produce a *horizontal* bar chart, with the bars in order of decreasing letter frequency (i.e., with the most common letter, E, at the bottom).

Problems

P7.1.1 *The Economist*'s Big Mac Index is a lighthearted measure of purchasing power parity between two currencies. Its premise is that the difference between the price of a McDonald's Big Mac hamburger in one currency (converted into US dollars (USD) at the prevailing exchange rate) and its price in the United States is a measure of the extent to which that currency is over- or under-valued (relative to the dollar).

The files at scipython.com/ex/aga provide the historical Big Mac prices and exchange rates for four currencies. For each currency, calculate the percentage over- or under-valuation of each currency as

$$\frac{(\text{local price converted to USD} - \text{US price})}{(\text{US price})} \times 100$$

and plot it as a function of time.

P7.1.2 Plot, as a histogram, the data in the table below concerning the number of cases of West Nile virus disease in the United States between 1999 and 2008. The two types of disease, neuroinvasive and non-neuroinvasive, should be plotted as separate bars on the same chart for each year.

Year	Neuroinvasive cases	Non-neuroinvasive cases
1999	59	3
2000	19	2
2001	64	2
2002	2,946	1,210
2003	2,866	6,996
2004	1,148	1,391
2005	1,309	1,691
2006	1,495	2,774
2007	1,227	117
2008	689	667

P7.1.3 A bubble chart is a type of scatterplot that can depict three dimensions of data through the position (x- and y-coordinates) and size of the marker. The `plt.scatter` method can produce bubble charts by passing the marker size to its `s` attribute (in $(\text{points})^2$ such that the area of the marker is proportional to the magnitude of the third dimension).

The files `gdp.tsv`, `bmi_men.tsv` and `population_total.tsv`, available at scipython.com/ex/agc, contain the following data from 2007 for each country: the GDP per person per capita in international dollars fixed at 2005 prices, the body mass index (BMI) of men (in kg/m^2) and the total population. Generate a bubble chart of BMI against GDP, in which the population is depicted by the size of the bubble markers. Beware: some data are missing for some countries.

Bonus exercise: color the bubbles by continent using the list provided in the file `continents.tsv`.

P7.1.4 The US National Oceanic and Atmospheric Administration (NOAA) makes a data set of atmospheric carbon dioxide concentrations since 1958 freely available to the public at ftp://aftp.cmdl.noaa.gov/products/trends/co2/co2_mm_mlo.txt. Using this data, plot the "interpolated" and "trend" CO_2 concentration against time on the same graph.

P7.1.5 Write a program to plot the Planck function, $B(\lambda)$, for the spectral radiance of a Black body at temperature T as a function of wavelength, λ for the Sun ($T = 5778$ K):

$$B(\lambda) = \frac{2hc^2}{\lambda^5}\frac{1}{\exp\left(hc/\lambda k_\mathrm{B}T\right) - 1}$$

Use a NumPy array to store values of $B(\lambda)$ from 100 to 5,000 nm, but set the wavelength range to *decrease* from 4,000 nm to 0.

P7.1.6 Reproduce Figure 7.19 using `Circle` patches.

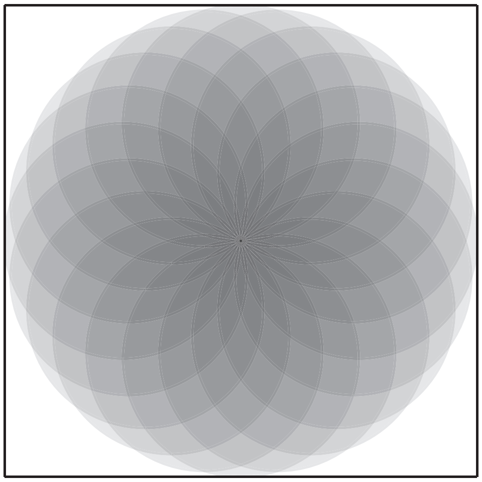


Figure 7.19 An image produced using Matplotlib `Circle` patches.

7.2 Contour plots, heatmaps and 3D plots

Until now, we have looked only at plotting one-dimensional data (that is, functions of one coordinate only). Matplotlib also supports several ways to plot data that is a function of two dimensions.

7.2.1 Contour plots

The `pyplot` method `contour` makes a contour plot of a provided two-dimensional array. In its simplest invocation, `contour(Z)`, no further arguments are required: the (x, y) values are indexes into the two-dimensional array `Z` and contour intervals are selected automatically. To explicitly include (x, y) coordinates, pass them as `contour(X, Y, Z)`. The arrays `X` and `Y` must have the same shape as `Z` (for example, as produced by `np.meshgrid`: see Section 6.1.6) or be one-dimensional such that `X` has the same length as the number of *columns* in `Z`, and `Y` has the same length as the number of *rows* in `Z`.

The contour levels can be controlled by a further argument: either a scalar, `N`, giving the total number of contour levels, or a sequence, `V`, explicitly listing the values of `Z` at which to draw contours.

The contours are colored according to Matplotlib's default *colormap*. In this process, the data are normalized linearly onto the interval `[0,1]`, which is then mapped onto a list of colors that are used to style the contours at the corresponding values. The module `matplotlib.cm` provides several colormap schemes:[14] some of the more practical ones are `cm.hot`, `cm.bone`, `cm.winter`, `cm.jet`, `cm.Greys` and `cm.hsv`. If you want to use a colormap with its colors reversed, tack a `_r` on the end of its name (e.g,, `cm.hot_r`).

[14] See the page http://wiki.scipy.org/Cookbook/Matplotlib/Show_colormaps for a complete list.

As an alternative, `contour` supports the `colors` argument which takes either a single Matplotlib color specifier or a sequence of such specifiers. For single-color contour plots, contours corresponding to negative values are plotted in dashed lines. The widths of the contour lines can be styled individually or all together with the argument `linewidths`.

Example E7.18 The following code produces a plot of the electrostatic potential of an electric dipole $\mathbf{p} = (qd, 0, 0)$ in the (x, y) plane for $q = 1.602 \times 10^{-19}$ C, $d = 1$ pm using the point dipole approximation (see Figure 7.20).

Listing 7.18 The electrostatic potential of a point dipole

```
# eg7-elec-dipole-pot.py
import numpy as np
import matplotlib.pyplot as plt

# Dipole charge (C), Permittivity of free space (F.m-1)
q, eps0 = 1.602e-19, 8.854e-12
# Dipole +q, -q distance (m) and a convenient combination of parameters
d = 1.e-12
k = 1/4/np.pi/eps0 * q * d

# Cartesian axis system with origin at the dipole (m)
X = np.linspace(-5e-11, 5e-11, 1000)
Y = X.copy()
X, Y = np.meshgrid(X, Y)

# Dipole electrostatic potential (V), using point dipole approximation
Phi = k * X / np.hypot(X, Y)**3

fig = plt.figure()
ax = fig.add_subplot(111)
# Draw contours at values of Phi given by levels
levels = np.array([10**pw for pw in np.linspace(0,5,20)])
levels = list(-levels) + list(levels)
# Monochrome plot of potential
ax.contour(X, Y, Phi, levels=levels, colors='k', linewidths=2)
plt.show()
```

To add labels to the contours, store the `ContourSet` object returned by the call to `ax.contour` and pass it to `ax.clabel` (perhaps with some additional parameters dictating the font properties). A further method, `ax.contourf`, which takes the same arguments as `contour`, draws *filled* contours. `contour` and `ax.contourf` can be used together, as in the following example.

Example E7.19 This program produces a filled contour plot of a function, labels the contours and provides some custom styling for their colors (see Figure 7.21).

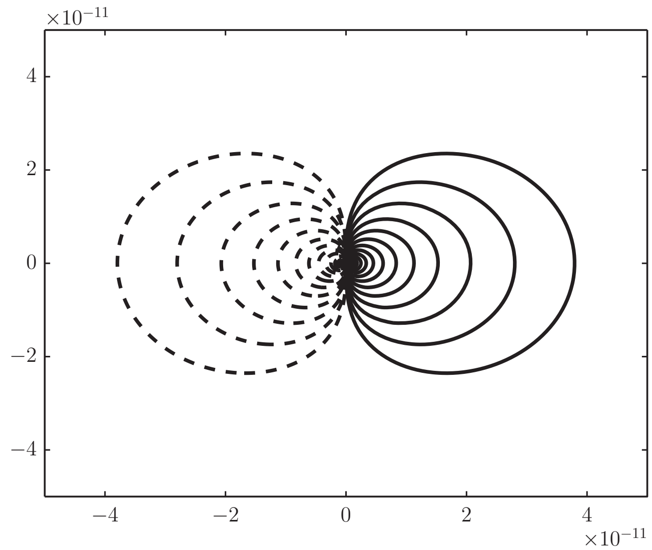


Figure 7.20 A contour plot of the electrostatic potential of a point dipole.

Listing 7.19 An example of filled and styled contours

```
# eg7-2dgau.py
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.cm as cm

X = np.linspace(0,1,100)
Y = X.copy()
X, Y = np.meshgrid(X, Y)
alpha = np.radians(25)
cX, cY = 0.5, 0.5
sigX, sigY = 0.2, 0.3
rX = np.cos(alpha) * (X-cX) - np.sin(alpha) * (Y-cY) + cX
rY = np.sin(alpha) * (X-cX) + np.cos(alpha) * (Y-cY) + cY

Z = (rX-cX)*np.exp(-((rX-cX)/sigX)**2) * np.exp(- ((rY-cY)/sigY)**2)
fig = plt.figure()
ax = fig.add_subplot(111)

# Reversed Greys colormap for filled contours
cpf = ax.contourf(X,Y,Z, 20, cmap=cm.Greys_r)
# Set the colors of the contours and labels so they're white where the
# contour fill is dark (Z < 0) and black where it's light (Z >= 0)
colors = ['w' if level<0 else 'k' for level in cpf.levels]
cp = ax.contour(X, Y, Z, 20, colors=colors)
ax.clabel(cp, fontsize=12, colors=colors)
plt.show()
```

7.2.2 Heatmaps

Another way to depict two-dimensional data is as a *heatmap*: an image in which the color of each pixel is determined by the corresponding value in the array of data. The use

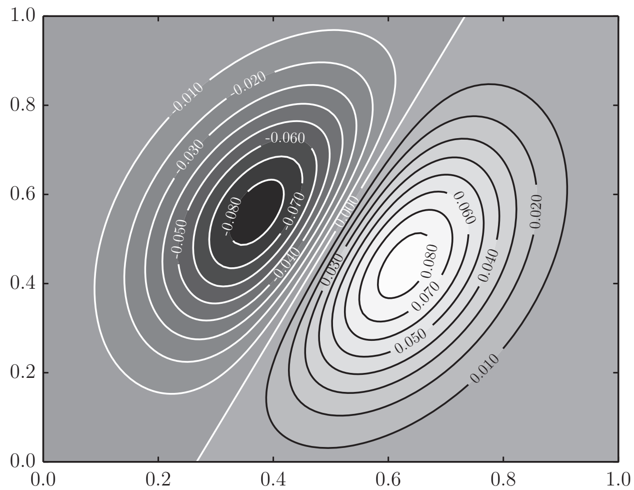


Figure 7.21 A two-dimensional plot with labeled contours.

of Matplotlib's functions, `ax.imshow`, `ax.pcolor` and `ax.pcolormesh` is described in this section.

`ax.imshow`

The `Axes` method `ax.imshow` displays an image on the axes. In its basic usage, it takes a two-dimensional array and maps its values to the pixels on an image according to some interpolation scheme and normalization. If the array data are taken from an image read in with the Matplotlib method `image.imread`, this is usually all that is required:

```
In [x]: import matplotlib.pyplot as plt
In [x]: import matplotlib.image as mpimg
In [x]: im = mpimg.imread('image.jpg')
In [x]: plt.imshow(im)
In [x]: plt.show()
```

(In this case, `im` is a three-dimensional array of shape `(n, m, 3)` in which the "depth" coordinate corresponds to the red, green and blue components of each pixel in the `n`-by-`m` image.)

`imshow` is frequently used to visualize matrices or other two-dimensional arrays of data. The default interpolation produces somewhat blurry-looking images for small arrays; for example, to visualize a 10×10 matrix as a 100×100 pixel image, a lot of intermediate points need to be approximated. To display the image with no interpolation, set `interpolation='none'` or `interpolation='nearest'`, as shown in the following example. Note that `imshow` takes a `cmap` argument that assigns a colormap in the same way as it does for `ax.contourf`.

Example E7.20 The following code compares two interpolation schemes: `'bilinear'`, which is the default on many new installations of Matplotlib and for a small array is blurry and `'nearest'`, which should look "blocky" (i.e., more faithful to the data): see Figure 7.22.

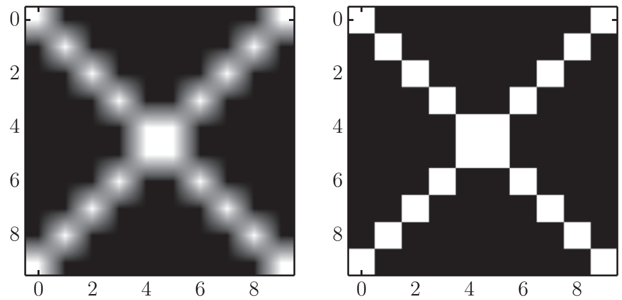


Figure 7.22 A small matrix visualized using `ax.imshow` with two different interpolation schemes.

Listing 7.20 A comparison of interpolation schemes for a small array visualized with `imshow()`

```
# eg7-matrix-show.py
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.cm as cm

# Make an array with ones in the shape of an 'X'
a = np.eye(10,10)
a += a[::-1,:]

fig = plt.figure()
ax1 = fig.add_subplot(121)
# Bilinear interpolation - this will look blurry
ax1.imshow(a, cmap=cm.Greys_r)

ax2 = fig.add_subplot(122)
# 'nearest' interpolation - faithful but blocky
ax2.imshow(a, interpolation='nearest', cmap=cm.Greys_r)

plt.show()
```

Example E7.21 The *Barnsley Fern* is a fractal that resembles the Black Spleenwort species of fern. It is constructed by plotting a sequence of points in the (x, y) plane, starting at $(0, 0)$, generated by the following affine transformations f_1, f_2, f_3, and f_4 where each transformation is applied to the previous point and chosen at random with probabilities $p_1 = 0.01$, $p_2 = 0.85$, $p_3 = 0.07$ and $p_4 = 0.07$.

$$f_1(x, y) = \begin{pmatrix} 0 & 0 \\ 0 & 0.16 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix}$$

$$f_2(x, y) = \begin{pmatrix} 0.85 & 0.04 \\ -0.04 & 0.85 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} 0 \\ 1.6 \end{pmatrix}$$

$$f_3(x, y) = \begin{pmatrix} 0.2 & -0.26 \\ 0.23 & 0.22 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} 0 \\ 1.6 \end{pmatrix}$$

$$f_4(x, y) = \begin{pmatrix} -0.15 & 0.28 \\ 0.26 & 0.24 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} 0 \\ 0.44 \end{pmatrix}$$

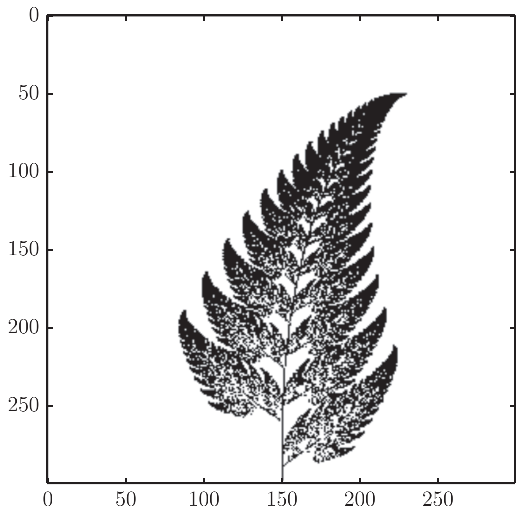


Figure 7.23 The Barnsley fern fractal.

This algorithm is implemented in the program below and the result is depicted in Figure 7.23.

Listing 7.21 Barnsley's fern

```
# eg7-fern.py
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.cm as cm

f1 = lambda x,y: (0., 0.16*y)
f2 = lambda x,y: (0.85*x + 0.04*y, -0.04*x + 0.85*y + 1.6)
f3 = lambda x,y: (0.2*x - 0.26*y, 0.23*x + 0.22*y + 1.6)
f4 = lambda x,y: (-0.15*x + 0.28*y, 0.26*x + 0.24*y + 0.44)
fs = [f1, f2, f3, f4]

npts = 50000
# Canvas size (pixels)
width, height = 300, 300
aimg = np.zeros((width, height))

x, y = 0, 0
for i in range(npts):
    # Pick a random transformation and apply it
    f = np.random.choice(fs, p=[0.01, 0.85, 0.07, 0.07])
    x, y = f(x,y)
    # Map (x,y) to pixel coordinates.
    # NB we "know" that -2.2 < x < 2.7 and 0 <= y < 10

    ix, iy = width / 2 + x * width / 10, y * height / 12
    # Set this point of the array to 1 to mark a point in the fern
    aimg[iy, ix] = 1

plt.imshow(aimg[::-1,:], cmap=cm.Greens)
plt.show()
```

`ax.pcolor` and `ax.pcolormesh`

There are a couple of other similar Matplotlib methods that you will come across: `ax.pcolor` and `ax.pcolormesh`. These are very similar. The precise differences are beyond the scope of this book, but `pcolormesh` is very much faster than `pcolor` and is the recommended alternative to `imshow` for this reason. The most noticeable difference is that `imshow` follows the convention used in the image-processing community that places the origin in the *top-left* corner; the `pcolor` methods associate the origin with the bottom-left corner.

Color bars

It is often useful to have a legend indicating how the colors of the plot relate to the values of the array used to derive it. This is added with the `fig.colorbar` method. In its most simple usage, simply call `fig.colorbar(`*`mappable`*`)` where *`mappable`* is the `Image`, `ContourSet` or other suitable object to which the colorbar applies and a new `Axes` object holding the colorbar will be created (and room made in the figure to accommodate it). This object can be further customized and labeled, as shown in the following examples.

Example E7.22 The following code reads in a data file of maximum daily temperatures in Boston for 2012 and plots them on a heatmap, with a labeled colorbar legend (see Figure 7.24). The data file may be downloaded from scipython.com/eg/aah.

Listing 7.22 Heatmap of Boston's temperatures in 2012

```
# eg7-heatmap.py

import numpy as np
import matplotlib.pyplot as plt

# Read in the relevant data from our input file
dt = np.dtype([('month', np.int), ('day', np.int), ('T', np.float)])
data = np.genfromtxt('boston2012.dat', dtype=dt, usecols=(1,2,3),
                     delimiter=(4,2,2,6))

# In our heatmap, nan will mean "no such date", e.g., 31 June
heatmap = np.empty((12, 31))
heatmap[:] = np.nan

for month, day, T in data:
    # NumPy arrays are zero-indexed; days and months are not!
    heatmap[month-1, day-1] = T

# Plot the heatmap, customize and label the ticks
fig = plt.figure()
ax = fig.add_subplot(111)
im = ax.imshow(heatmap, interpolation='nearest')
ax.set_yticks(range(12))
ax.set_yticklabels(['Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun',
                    'Jul', 'Aug', 'Sep', 'Oct', 'Nov', 'Dec'])
days = np.array(range(0, 31, 2))
ax.set_xticks(days)
```

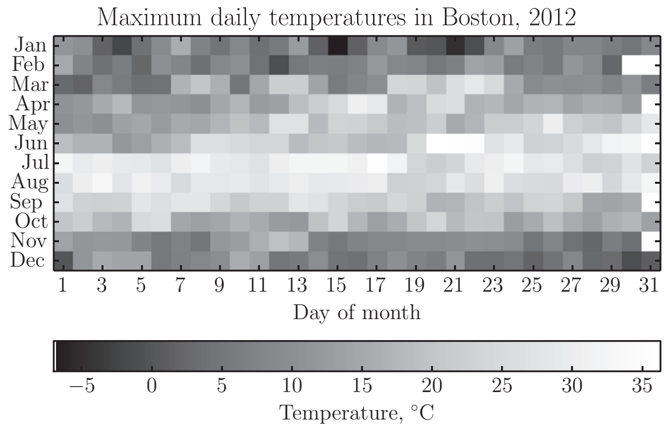


Figure 7.24 A heatmap of maximum daily temperatures in Boston during 2012.

```
ax.set_xticklabels(['{:d}'.format(day+1) for day in days])
ax.set_xlabel('Day of month')
ax.set_title('Maximum daily temperatures in Boston, 2012')

# Add a colorbar along the bottom and label it
cbar = fig.colorbar(ax=ax, mappable=im, orientation='horizontal')  ❶
cbar.set_label('Temperature, $^\circ\mathrm{C}$')

plt.show()
```

❶ The "mappable" object passed to `fig.colorbar` is the `AxesImage` object returned by `ax.imshow`.

Example E7.23 The two-dimensional diffusion equation is

$$\frac{\partial U}{\partial t} = D\left(\frac{\partial^2 U}{\partial x^2} + \frac{\partial^2 U}{\partial y^2}\right)$$

where D is the diffusion coefficient. A simple numerical solution on the domain of the unit square $0 \le x < 1, 0 \le y < 1$ approximates $U(x, y; t)$ by the discrete function $u_{i,j}^{(n)}$ where $x = i\Delta x$, $y = j\Delta y$ and $t = n\Delta t$. Applying finite difference approximations yields

$$\frac{u_{i,j}^{(n+1)} - u_{i,j}^{(n)}}{\Delta t} = D\left[\frac{u_{i+1,j}^{(n)} - 2u_{i,j}^{(n)} + u_{i-1,j}^{(n)}}{(\Delta x)^2} + \frac{u_{i,j+1}^{(n)} - 2u_{i,j}^{(n)} + u_{i,j-1}^{(n)}}{(\Delta y)^2}\right],$$

and hence the state of the system at time step $n+1$, $u_{i,j}^{(n+1)}$ may be calculated from its state at time step n, $u_{i,j}^{(n)}$ through the equation

$$u_{i,j}^{(n+1)} = u_{i,j}^{(n)} + D\Delta t\left[\frac{u_{i+1,j}^{(n)} - 2u_{i,j}^{(n)} + u_{i-1,j}^{(n)}}{(\Delta x)^2} + \frac{u_{i,j+1}^{(n)} - 2u_{i,j}^{(n)} + u_{i,j-1}^{(n)}}{(\Delta y)^2}\right].$$

Consider the diffusion equation applied to a metal plate initially at temperature T_{cold} apart from a disc of a specified size, which is at temperature T_{hot}. We suppose that the edges of the plate are held fixed at T_{cool}. The following code applies the above formula to follow the evolution of the temperature of the plate. It can be shown that the maximum time step, Δt, that we can allow without the process becoming unstable is

$$\Delta t = \frac{1}{2D}\frac{(\Delta x \Delta y)^2}{(\Delta x)^2 + (\Delta y)^2}.$$

In the code below, each call to `do_timestep` updates the numpy array `u` from the results of the previous time step, `u0`. The simplest approach to applying the partial difference equation is to use a Python loop:

```
for i in range(1, nx-1):
    for j in range(1, ny-1):
        uxx = (u0[i+1,j] - 2*u0[i,j] + u0[i-1,j]) / dx2
        uyy = (u0[i,j+1] - 2*u0[i,j] + u0[i,j-1]) / dy2
        u[i,j] = u0[i,j] + dt * D * (uxx + uyy)
```

However, this runs extremely slowly and using vectorization will farm out these explicit loops to the much faster precompiled C code underlying NumPy's array implementation.

The state of the system is plotted as an image at four different stages of its evolution (see Figure 7.25).

Listing 7.23 The two-dimensional diffusion equation applied to the temperature of a steel plate

```
# eg7-diffusion2d.py
import numpy as np
import matplotlib.pyplot as plt

# plate size, mm
w = h = 10.
# intervals in x-, y- directions, mm
dx = dy = 0.1
# Thermal diffusivity of steel, mm2.s-1
D = 4.

Tcool, Thot = 300, 700

nx, ny = int(w/dx), int(h/dy)

dx2, dy2 = dx*dx, dy*dy
dt = dx2 * dx2 / (2 * D * (dx2 + dy2))

u0 = Tcool * np.ones((nx, ny))
u = np.empty((nx, ny))

# Initial conditions - ring of inner radius r, width dr centered at (cx,cy) (mm)
r, cx, cy = 2, 5, 5
r2 = r**2
for i in range(nx):
    for j in range(ny):
        p2 = (i*dx-cx)**2 + (j*dy-cy)**2
        if p2 < r2:
            u0[i,j] = Thot
```

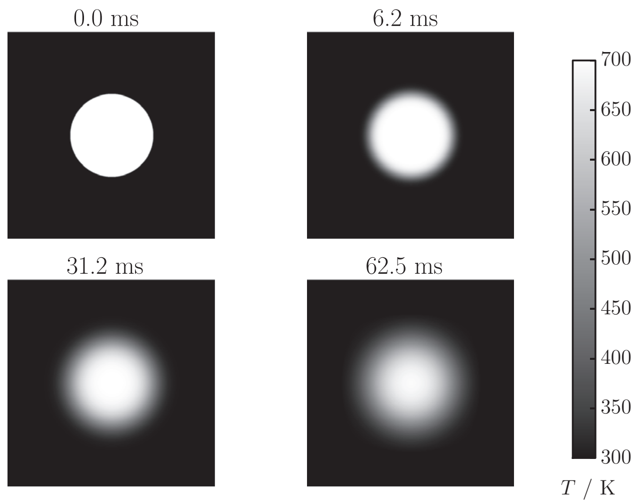


Figure 7.25 A representation of the temperature of a circular disc at four times after its instantaneous heating.

```
def do_timestep(u0, u):
    # Propagate with forward-difference in time, central-difference in space
    u[1:-1, 1:-1] = u0[1:-1, 1:-1] + D * dt * (
          (u0[2:, 1:-1] - 2*u0[1:-1, 1:-1] + u0[:-2, 1:-1])/dx2
          + (u0[1:-1, 2:] - 2*u0[1:-1, 1:-1] + u0[1:-1, :-2])/dy2 )

    u0 = u.copy()
    return u0, u

# Number of timesteps
nsteps = 101
# Output 4 figures at these timesteps
mfig = [0, 10, 50, 100]
fignum = 0
fig = plt.figure()
for m in range(nsteps):
    u0, u = do_timestep(u0, u)
    if m in mfig:
        fignum += 1
        print(m, fignum)
        ax = fig.add_subplot(220 + fignum)
        im = ax.imshow(u.copy(), cmap=plt.get_cmap('hot'), vmin=Tcool,vmax=Thot)
        ax.set_axis_off()
        ax.set_title('{:.1f} ms'.format(m*dt*1000))
fig.subplots_adjust(right=0.85)
cbar_ax = fig.add_axes([0.9, 0.15, 0.03, 0.7])   ❶
cbar_ax.set_xlabel('$T$ / K', labelpad=20)
fig.colorbar(im, cax=cbar_ax)
plt.show()
```

❶ To set a common `colorbar` for the four plots we define its own `Axes`, `cbar_ax` and make room for it with `fig.subplots_adjust`. The plots all use the same color range, defined by `vmin` and `vmax`, so it doesn't matter which one we pass in the first argument to `fig.colorbar`.

7.2.3 3D plots

Matplotlib is primarily a 2D plotting library, but it does support 3D plotting functionality that is good enough for many purposes. The easiest way to set up a 3D plot is to import `Axes3D` from the `mpl_toolkits.mplot3d` module and to set the subplot's `projection` argument to `'3d'`:

```
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
```

The corresponding `Axes` object can then depict data in three dimensions as a line plot, scatterplot, wireframe plot or surface plot.[15]

`ax.plot_wireframe` and `ax.plot_surface`

The simplest kind of surface plot is a wireframe plot that draws lines in 3D perspective joining the provided two-dimensional array of points, `Z`, on a grid of data values provided as two-dimensional arrays `X` and `Y` (as for `imshow` and `contour`). By default, wires are drawn for every point in the array: if this is too many, set the arguments `rstride` and `cstride` to specify the array row step size and column step size.

The `ax.plot_surface` method is similar but produces a surface plot of filled patches. The patch colors can be set to a single color with the `color` argument or styled to a specifed color map with the `cmap` argument. `rstride` and `cstride` default to 10 for the `ax.plot_surface` method. Both methods are illustrated in the following example.

Example E7.24 Some of the different options for producing surface plots are illustrated by the code below, which produces Figure 7.26.

Listing 7.24 Four 3D plots of a simple two-dimensional Gaussian function

```
# eg7-3d-surface-plots.py
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.cm as cm

L, n = 2, 400
x = np.linspace(-L, L, n)
y = x.copy()
X, Y = np.meshgrid(x, y)
Z = np.exp(-(X**2 + Y**2))

fig, ax = plt.subplots(nrows=2, ncols=2, subplot_kw={'projection': '3d'})
ax[0,0].plot_wireframe(X, Y, Z, rstride=40, cstride=40)
ax[0,1].plot_surface(X, Y, Z, rstride=40, cstride=40, color='m')
```

[15] It is even possible to produce three-dimensional contour plots and bar charts, though these are of doubtful use in practice.

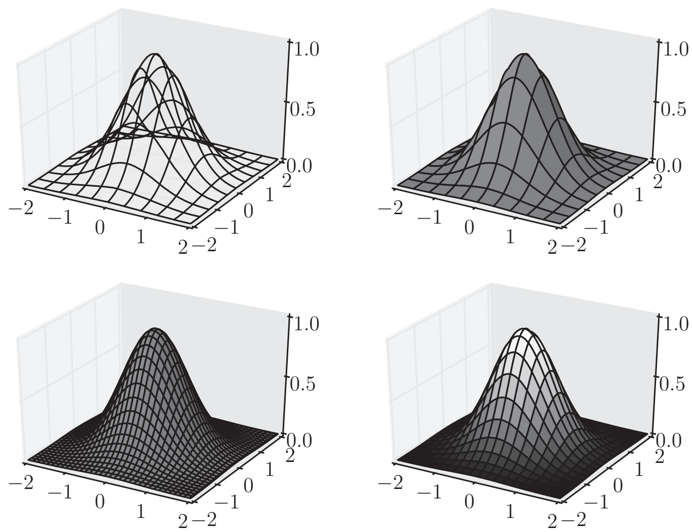


Figure 7.26 Four different 3D surface plots of the same function.

```
ax[1,0].plot_surface(X, Y, Z, rstride=12, cstride=12, color='m')
ax[1,1].plot_surface(X, Y, Z, rstride=20, cstride=20, cmap=cm.hot)
for axes in ax.flatten():
    axes.set_xticks([-2, -1, 0, 1, 2])
    axes.set_yticks([-2, -1, 0, 1, 2])
    axes.set_zticks([0, 0.5, 1])
fig.tight_layout()
plt.show()
```

In an interactive plot, the viewing direction can be changed by clicking and dragging on the plot. To fix a particular viewing direction for a static plot image, pass the required elevation and azimuthal angles (in degrees, in that order) to `ax.view_init`, as in the following example.

Example E7.25 The parametric description of a torus with radius c and tube radius a is

$$x = (c + a\cos\theta)\cos\phi$$
$$y = (c + a\cos\theta)\sin\phi$$
$$z = a\sin\theta$$

for θ and ϕ each between 0 and 2π. The code below outputs two views of a torus rendered as a surface plot (Figure 7.27).

Listing 7.25 A 3D surface plot of a torus

```
# eg7-torus-surface.py
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
```

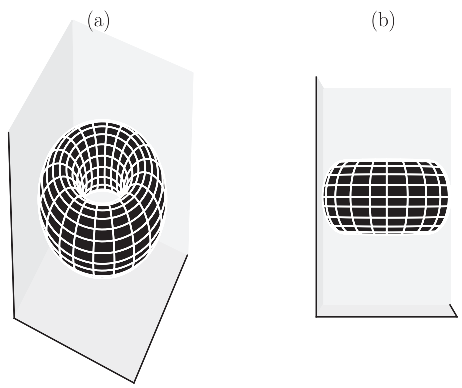


Figure 7.27 Two views of the same torus: (a) $\theta = 36°, \phi = 26°$, (b) $\theta = 0°, \phi = 0°$.

```
n = 100

theta = np.linspace(0, 2.*np.pi, n)
phi = np.linspace(0, 2.*np.pi, n)
❶ theta, phi = np.meshgrid(theta, phi)
c, a = 2, 1
x = (c + a*np.cos(theta)) * np.cos(phi)
y = (c + a*np.cos(theta)) * np.sin(phi)
z = a * np.sin(theta)

fig = plt.figure()
ax1 = fig.add_subplot(121, projection='3d')
ax1.set_zlim(-3,3)
❷ ax1.plot_surface(x, y, z, rstride=5, cstride=5, color='k', edgecolors='w')
❸ ax1.view_init(36, 26)
ax2 = fig.add_subplot(122, projection='3d')
ax2.set_zlim(-3,3)
ax2.plot_surface(x, y, z, rstride=5, cstride=5, color='k', edgecolors='w')
ax2.view_init(0, 0)
ax2.set_xticks([])
plt.show()
```

❶ We need θ and ϕ to range over the interval $(0, 2\pi)$ independently, so we use a `meshgrid`.

❷ Note that we can use keywords such as `edgecolors` to style the polygon patches created by `ax.plot_surface`.

❸ Elevation angle above the *xy*-plane of 36°, azimuthal angle in the *xy*-plane of 26°.

Line plots and scatterplots

Line plots and scatterplots work in 3D in a way similar to how they work in 2D: the basic method call is `ax.plot(x, y, z)` and `ax.scatter(x, y, z)`, where `x`, `y` and `z` are equal-length, one-dimensional arrays. Only limited annotation of such plots is possible without using advanced methods, however.

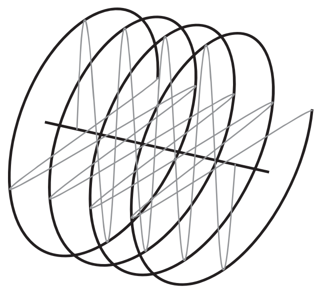


Figure 7.28 A depiction of circularly polarized light as a helix on a three-dimensional plot.

Example E7.26 Below is a simple example of a three-dimensional plot of a helix, which could represent circularly polarized light, for example. See Figure 7.28.

Listing 7.26 A depiction of a helix on a three-dimensional plot

```
# eg7-circular-polarization.py
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

n = 1000
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')

# Plot a helix along the x-axis
theta_max = 8 * np.pi
theta = np.linspace(0, theta_max, n)
x = theta
z =  np.sin(theta)
y =  np.cos(theta)
ax.plot(x, y, z, 'b', lw=2)

# An line through the center of the helix
ax.plot((-theta_max*0.2, theta_max * 1.2), (0,0), (0,0), color='k', lw=2)
# sin/cos components of the helix (e.g., electric and magnetic field
# components of a circularly polarized electromagnetic wave
ax.plot(x, y, 0, color='r', lw=1, alpha=0.5)
ax.plot(x, [0]*n, z, color='m', lw=1, alpha=0.5)

# Remove axis planes, ticks and labels
ax.set_axis_off()
plt.show()
```

7.2.4 Exercises

Questions

Q7.2.1 Generate an image plot of the sinc function in the Cartesian plane, $\text{sinc}(r) = \sin r/r$ where $r = \sqrt{x^2 + y^2}$.

Q7.2.2 The data provided in the comma-separated file `birthday-data.csv`, available at scipython.com/ex/agd gives the number of births recorded by the US Centers for Disease Control and Prevention's National Center for Health Statistics for each day of the year as a total from years 1969–1988. The columns are month number (1=January, 12=December), day number and number of live births.

Use NumPy to estimate, for each day of the year, the probability of a particular individual's birthday being on that day. Plot the probabilities as a heatmap like that of Example E7.22 and investigate any features of interest.

Hint: the data need "cleaning" to a small extent – inspect the data file first to establish the presence of any incorrect entries.

Problems

P7.2.1 The so-called *'chaos game'* is an algorithm for generating a fractal. First define the n vertices of a regular polygon and an initial point, (x_0, y_0) selected at random within the polygon. Then generate a sequence of points, starting with (x_0, y_0), where each point is a fraction r of the distance between the previous one and a polygon vertex chosen at random. For example, the algorithm applied with parameters $n = 3, r = 0.5$ generates a Sierpinski triangle.

Write a program to draw fractals using the chaos game algorithm.

P7.2.2 Extend the code in Example E7.16 to include contours of body mass index, defined by BMI = (mass/kg)/(height/m)^2. Plot these contours to delimit the supposed categories of "under-weight" (< 18.5), "over-weight" (> 25) and "obese" (> 30). Manually place the contour labels so that they are out of the way of the scatterplotted data points and format them to one decimal place.

P7.2.3 The two-dimensional *advection equation* may be written

$$\frac{\partial U}{\partial t} = -v_x \frac{\partial U}{\partial x} - v_y \frac{\partial U}{\partial y},$$

where $\mathbf{v} = (v_x, v_y)$ is the vector velocity field (giving the velocity components v_x and v_y, which may vary as a function of position, (x, y)). In a similar way to the approach taken in Example E7.23, this equation may be discretized and solved numerically. With forward-differences in time and central-differences in space, we have

$$u_{i,j}^{(n+1)} = u_{i,j}^{(n)} - \Delta t \left[v_{x;i,j} \frac{u_{i+1,j}^{(n)} - u_{i-1,j}^{(n)}}{2\Delta x} + v_{y;i,j} \frac{u_{i,j+1}^{(n)} - u_{i,j-1}^{(n)}}{2\Delta y} \right].$$

Implement this approximate numerical solution on the domain $0 \le x < 10, 0 \le y < 10$ discretized with $\Delta x = \Delta y = 0.1$ with the initial condition

$$u_0(x, y) = \exp\left(-\frac{(x - c_x)^2 + (y - c_y)^2}{\alpha^2} \right),$$

where $(c_x, c_y) = (5, 5)$ and $\alpha = 2$. Take the velocity field to be a circulation at constant speed 0.1 about an origin at $(7, 5)$.

P7.2.4 The *Julia set* associated with the complex function $f(z) = z^2 + c$ may be depicted using the following algorithm.

For each point, z_0, in the complex plane such that $-1.5 \le \mathrm{Re}[z_0] \le 1.5$ and $-1.5 \le \mathrm{Im}[z_0] \le 1.5$, iterate according to $z_{n+1} = z_n^2 + c$. Color the pixel in an image corresponding to this region of the complex plane according to the number of iterations required for $|z|$ to exceed some critical value, $|z|_{\max}$ (or black if this does not happen before a certain maxmimum number of iterations $n_{\max}$).

Write a program to plot the Julia set for $c = -0.1 + 0.65j$, using $|z|_{\max} = 10$ and $n_{\max} = 500$.

P7.2.5 The mean altitudes of the 10 km × 10 km *hectad* squares used by the UK's Ordnance Survey in mapping Great Britain are given in the NumPy array file `gb-alt.npy`, available at scipython.com/ex/agb. NaN values in this array denote the sea.

Plot a map of the island using this data with `ax.imshow` and plot further maps assuming a mean sea-level rise of (a) 10 m, (b) 50 m, (c) 200 m. In each case, deduce the percentage of land area remaining, relative to its present value.

8 SciPy

SciPy is a library of Python modules for scientific computing that provides more specific functionality than the generic data structures and mathematical algorithms of NumPy. For example, it contains modules for the evaluation of special functions frequently encountered in science and engineering, optimization, integration, interpolation and image manipulation. As with the NumPy library, many of SciPy's underlying algorithms are executed as compiled C code, so they are fast. Also like NumPy and Python itself, SciPy is free software.

There is little new syntax to learn in using the SciPy routines, so this chapter will focus on examples of the library's use in short programs of relevance to science and engineering.

8.1 Physical constants and special functions

The useful `scipy.constants` package provides the internationally agreed standard values and uncertainties for physical constants. The `scipy.special` package also supplies a large number of algorithms for calculating functions that appear in science, mathematical analysis and engineering, including:

- Airy functions
- Elliptic functions and integrals
- Bessel functions, their zeros, derivatives and integrals
- Spherical Bessel functions
- Struve functions
- A variety of statistical functions and distributions
- Gamma and beta functions
- The error function
- Fresnel integrals
- Legendre functions and associated Legendre functions
- A variety of orthogonal polynomials
- Hypergeometric functions
- Parabolic cylinder functions
- Matheiu functions
- Spheroidal functions
- Kelvin functions

They are described in detail in the documentation;[1] we focus in this section on a few representative examples.

Most of these special functions are implemented in SciPy as universal functions: that is, they support broadcasting and vectorization (automatic array-looping), and so work as expected with NumPy arrays.

8.1.1 Physical constants

SciPy contains the 2010 CODATA internationally recommended values[2] of many physical constants. They are held, with their units and uncertainties, in a dictionary, `scipy.constants.physical_constants`, keyed by an identifying string. For example,

```
In [x]: import scipy.constants as pc
In [x]: pc.physical_constants['Avogadro constant']
Out[x]: (6.02214129e+23, 'mol^-1', 2.7e+16)
```

The convenience methods `value`, `unit` and `precision` retrieve the corresponding properties on their own:

```
In [x]: pc.value('elementary charge')
Out[x]: 1.602176565e-19
In [x]: pc.unit('elementary charge')
Out[x]: 'C'
In [x]: pc.precision('elementary charge')
2.1845282701410628e-08
```

To save typing, it is usual to assign the value to a variable name at the start of a program, for example,

```
In [x]: muB = pc.value('Bohr magneton')
```

A full list of the constants and their names is given in the SciPy documentation,[3] but Table 8.1 lists the more important ones. Some particularly important constants have a direct variable assignment within `scipy.constants` (in SI units) and so can be imported directly:

```
In [x]: from scipy.constants import c, R, k
In [x]: c, R, k     # speed of light, gas constant, Boltzmann constant
Out[x]: (299792458.0, 8.3144621, 1.3806488e-23)
```

Where this is the case, the variable name is given in the table. You will probably find it convenient to use the `scipy.constants` values, but should be aware that if and when newer values are released the package may be updated – this means that your code may produce slightly different results for different versions of SciPy.

There are one or two useful conversion factors and methods, and SI prefixes defined within the `scipy.constants` package, for example,

---

[1] http://docs.scipy.org/doc/scipy/reference/special.html.

[2] P. J. Mohr, B. N. Taylor, D. B. Newell, (2012). *Rev. Mod. Phys.*, **84**, 1527.

[3] http://docs.scipy.org/doc/scipy/reference/constants.html.

Table 8.1 Physical constants in `scipy.constants`

Constant string	Variable	Value	Units
'atomic mass constant'		1.660538921e-27	kg
'Avogadro constant'	N_A	6.02214129e+23	mol^{-1}
'Bohr magneton'		9.27400968e-24	$J\,T^{-1}$
'Bohr radius'		5.2917721092e-11	m
'Boltzmann constant'	k	1.3806488e-23	$J\,K^{-1}$
'electron mass'	m_e	9.10938291e-31	kg
'elementary charge'	e	1.602176565e-19	C
'Faraday constant'		96485.3365	$C\,mol^{-1}$
'fine-structure constant'	alpha	0.0072973525698	
'molar gas constant'	R	8.3144621	$J\,K^{-1}\,mol^{-1}$
'neutron mass'	m_n	1.674927351e-27	kg
'Newtonian constant of gravitation'	G	6.67384e-11	$m^3\,kg^{-1}\,s^{-2}$
'Planck constant'	h	6.62606957e-34	J s
'Planck constant over 2 pi'	hbar	1.054571726e-34	J s
'proton mass'	m_p	1.672621777e-27	kg
'Rydberg constant'	Rydberg	10973731.5685	m^{-1}
'speed of light in vacuum'	c	299792458.0	$m\,s^{-1}$

```
In [x]: import scipy.constants as pc
In [x]: pc.atm
Out[x]: 101325.0              # 1 atm in Pa
In [x]: pc.bar
Out[x]: 100000.0              # 1 bar in Pa
In [x]: pc.torr
Out[x]: 133.32236842105263    # 1 torr in Pa
In [x]: pc.zero_Celsius
Out[x]: 273.15                # 0 degC in K
In [x]: pc.micro              # also nano, pico, mega, giga, etc.
Out[x]: 1e-06
```

Example E8.1 Let's use the `scipy.constants.physical_constants` dictionary to determine which are the least accurately known constants. To do this we need the *relative* uncertainties in the constants' values. The code mentioned here uses a structured array to calculate these and outputs the least well-determined constants.

Listing 8.1 Least well-defined physical constants

```
import numpy as np

from scipy.constants import physical_constants

def make_record(k, v):
    """
    Return the record for this constant from the key and value of its entry
    in the physical_constants dictionary.
```

```
    """
    name = k
    val, units, abs_unc = v
    # Calculate the relative uncertainty in ppm
    rel_unc = abs_unc / abs(val) * 1.e6
    return name, val, units, abs_unc, rel_unc

dtype = [('name', 'S50'), ('val', 'f8'), ('units', 'S20'),
         ('abs_unc', 'f8'), ('rel_unc', 'f8')]
constants = np.array([make_record(k, v) for k,v in physical_constants.items()],
                     dtype=dtype )
constants.sort(order='rel_unc')

# List the 10 constants with the largest relative uncertainties
for rec in constants[-10:]:
    print('{:.0f} ppm: {:s} = {:g} {:s}'.format(rec['rel_unc'],
          rec['name'].decode(), rec['val'], rec['units'].decode()))
```

The output is shown here. Note that *G* is not known to better than about 120 ppm (parts per million.)

```
91 ppm: proton-tau mass ratio = 0.528063
91 ppm: tau mass energy equivalent = 2.84678e-10 J
92 ppm: tau mass = 3.16747e-27 kg
119 ppm: Newtonian constant of gravitation over h-bar c = 6.70837e-39 (GeV/c^2)^-2
120 ppm: Newtonian constant of gravitation = 6.67384e-11 m^3 kg^-1 s^-2
545 ppm: proton mag. shielding correction = 2.5694e-05
545 ppm: proton magn. shielding correction = 2.5694e-05
980 ppm: deuteron rms charge radius = 2.1424e-15 m
5812 ppm: proton rms charge radius = 8.775e-16 m
9447 ppm: weak mixing angle = 0.2223
```

8.1.2 Airy and Bessel functions

The Airy functions Ai(x) and Bi(x) are the linearly independent solutions to the Airy equation, $y'' - xy = 0$, which occurs in quantum mechanics, optics, electrodynamics and other areas of physics. The functions (`Ai`, `Bi`) and their derivatives (`Aip`, `Bip`) are returned by the function `scipy.special.airy`. The only required argument is `x`, which could be complex and can be a NumPy array:

```
In [x]: Ai, Aip, Bi, Bip = airy(0)
In [x]: Ai, Aip, Bi, Bip
(0.35502805388781722, -0.25881940379280682, 0.61492662744600068, 0.44828835735382638)
```

The first `nt` zeros of the Airy functions and their derivatives are returned by the function `scipy.special.ai_zeros(nt)`:

```
In [x]: a, ap, ai, aip = ai_zeros(2)    # arrays for the first 2 zeros of Ai
In [x]: a[1], ap[1], ai[1], aip[1]      # look at the 2nd zero:
Out[x]: (-4.0879494441309721, -3.248197582179837, -0.41901547803256406,
         -0.80311136965486463)
In [x]: airy(a[1])[0]                   # Ai(a) should = 0
Out[x]: 1.2774882441379295e-15          # close enough
In [x]: airy(ap[1])[1]                  # Aip(ap) should = 0
```

```
Out[x]: -3.2322209157744908e-16          # close enough
In [x]: airy(ap[1])[0]                   # Ai(ap) is returned as ai above
Out[x]: -0.41901547803256395
In [x]: airy(a[1])[1]                    # Aip(a) is returned as aip above
Out[x]: -0.80311136965486396
```

◇ **Example E8.2** Consider a particle of mass m moving in a constant gravitational field such that its potential energy at a height z above a surface is mgz. If the particle bounces elastically on the surface, the classical probability density corresponding to its position is

$$P_{\mathrm{cl}}(z) = \frac{1}{\sqrt{z_{\mathrm{max}}(z_{\mathrm{max}} - z)}},$$

where z_{max} is the maximum height it reaches.

The quantum mechanical behaviour of this system may described by the solution to the time-independent Schrödinger equation,

$$-\frac{\hbar^2}{2m}\frac{\mathrm{d}^2\psi}{\mathrm{d}z^2} + mgz\psi = E\psi$$

which is simplified by the coordinate rescaling $q = z/\alpha$ where $\alpha = (\hbar^2/2m^2g)^{1/3}$:

$$\frac{\mathrm{d}^2\psi}{\mathrm{d}q^2} - (q - q_E)\psi = 0, \quad \text{where } q_E = \frac{E}{mg\alpha}.$$

The solutions to this differential equation are the Airy functions. The boundary condition $\psi(z) \to 0$ as $z \to \infty$ specifically gives:

$$\psi(q) = N_E\mathrm{Ai}(q - q_E),$$

where N_E is a normalization constant.

The second boundary condition, $\psi(q = 0) = 0$, leads to quantization in terms of a quantum number $n = 1, 2, 3, \cdots$ with scaled energy values q_E found from the zeros of the Airy function: $\mathrm{Ai}(-q_E) = 0$.

The following program plots the classical and quantum probability distributions, $P_{\mathrm{cl}}(z)$ and $|\psi(z)|^2$, for $n = 1$ and $n = 16$ (Figure 8.1).

Listing 8.2 Probability densities for a particle in a uniform gravitational field

```
# eg8-qm-gravfield.py
import numpy as np
from scipy.special import airy, ai_zeros
import pylab

nmax = 16

# Find the first nmax zeros of Ai(x)
❶ a, _, _, _ = ai_zeros(nmax)
# The actual boundary condition is Ai(-qE) = 0 at q=0, so:
qE = -a

def prob_qm(n):
```

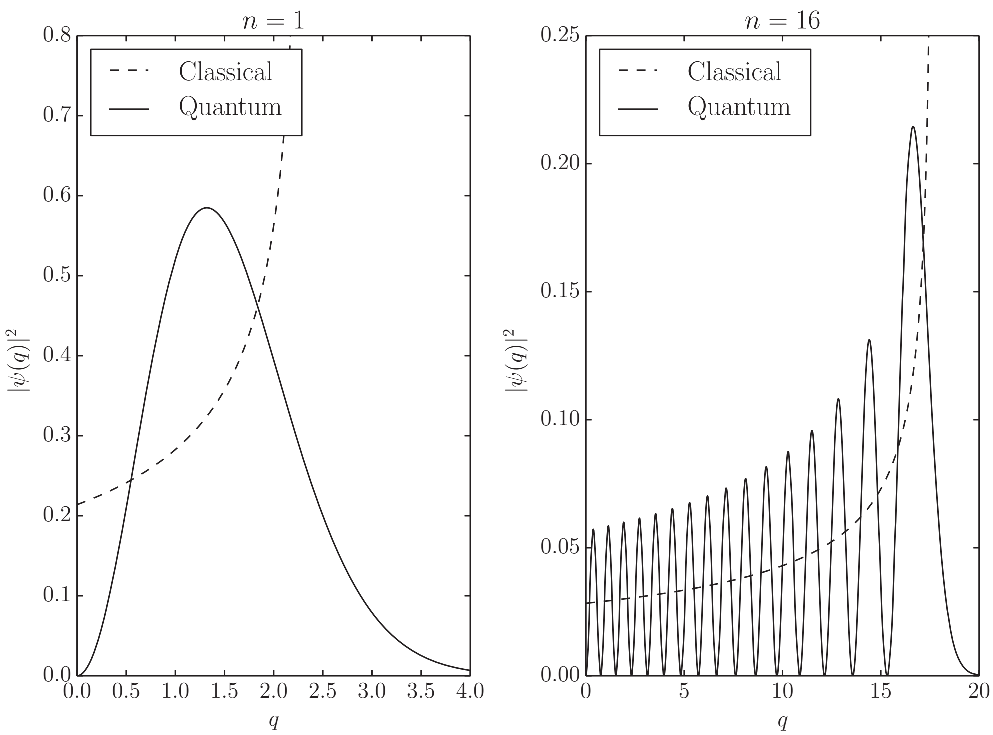


Figure 8.1 A comparison of classical and quantum probability distributions for a particle moving in a constant gravitationa field at two different energies.

```
    """
    Return the quantum mechanical probability density for a particle moving
    in a uniform gravitational field.

    """
    # The quantum mechanical wavefunction is proportional to Ai(q-qE) where
    # the qE corresponding to quantum number n is indexed at n-1
❷   psi, _, _, _ = airy(q-qE[n-1])
    # Return the probability density, after rough-and-ready normalization
    P = psi**2
❸   return P / (sum(P) * dq)

def prob_cl(n):
    """
    Return the classical probability density for a particle bouncing
    elastically in a uniform gravitational field.

    """
    # The classical probability density is already normalized
    return 0.5/np.sqrt(qE[n-1]*(qE[n-1]-q))

# The ground state, n=1
q, dq = np.linspace(0, 4, 1000, retstep=True)
pylab.plot(q, prob_cl(1), label='Classical')
pylab.plot(q, prob_qm(1), label='Quantum')
pylab.ylim(0,0.8)
pylab.legend()
pylab.show()
```

```
# An excited state, n=16
q, dq = np.linspace(0, 20, 1000, retstep=True)
pylab.plot(q, prob_cl(16), label='Classical')
pylab.plot(q, prob_qm(16), label='Quantum')
pylab.ylim(0,0.25)
pylab.legend(loc='upper left')
pylab.show()
```

❶ We use `scipy.special.ai_zeros` to retrieve the $n = 1$ and $n = 16$ eigenvalues.

❷ `scipy.special.airy` finds the corresponding wavefunctions and hence probability densities.

❸ For the sake of illustration, these are normalized approximately by a very simple numerical integration.

Bessel functions are another important class of function with many applications to physics and engineering. SciPy provides several functions for evaluating them, their derivatives and their zeros.

- `jn(v, x)` and `jv(v, x)` return the *Bessel function of the first kind* at `x` for order `v`, $J_\nu(x)$. `v` can be real or integer.
- `yn(n, x)` and `yv(v, x)` return the *Bessel function of the second kind* at `x` for integer order `n` ($Y_n(x)$) and real order `v` ($Y_\nu(x)$), respectively.
- `in(n, x)` and `iv(v, x)` return the *modified Bessel function of the first kind* at `x` for integer order `n` ($I_n(x)$) and real order `v` ($I_\nu(x)$), respectively.
- `kn(n, x)` and `kv(v, x)` return the *modified Bessel function of the second kind* at `x` for integer order `n` ($K_n(x)$) and real order `v` ($K_\nu(x)$), respectively.
- The functions `jvp(v, x)`, `yvp(v, x)`, `ivp(v, x)` and `kvp(v, x)` return the *derivatives* of the earlier mentioned functions. By default, the first derivative is returned; to return the nth derivative, set the optional argument, `n`.
- Several functions can be used to obtain the *zeros* of the Bessel functions. Probably the most useful are `jn_zeros(n, nt)`, `jn_zeros(n, nt)`, `jnp_zeros(n, nt)`, `yn_zeros(n, nt)` and `ynp_zeros(n, nt)`, which return the first `nt` zeros of $J_n(x)$, $J'_n(x)$, $Y_n(x)$ and $Y'_n(x)$.

Example E8.3 The vibrations of a thin circular membrane stretched across a rigid circular frame (such as a drum head) can be described as normal modes written in terms of Bessel functions:

$$z(r, \theta; t) = AJ_n(kr) \sin n\theta \cos k\nu t,$$

where (r, θ) describes a position in polar coordinates with the origin at the center of the membrane, t is time and v is a constant depending on the tension and surface density of the drum. The modes are labeled by integers $n = 0, 1, \cdots$ and $m = 1, 2, 3, \cdots$ where k is the mth zero of J_n.

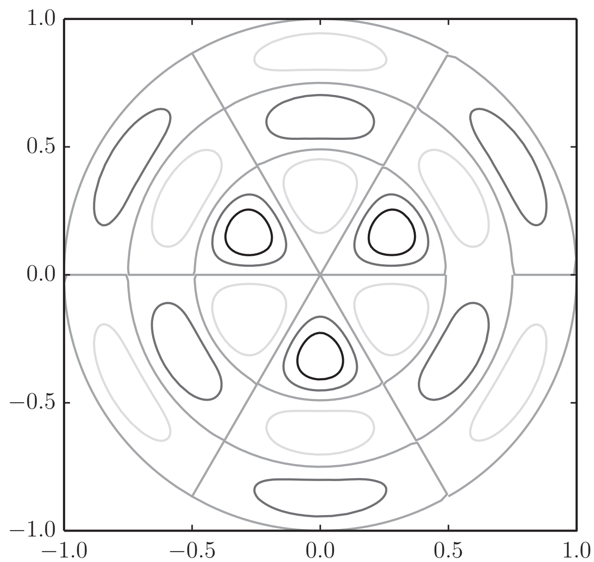


Figure 8.2 The $n = 3, m = 2$ normal mode of a vibrating circular drum.

The following program produces a plot of the displacement of the membrane in the $n = 3, m = 2$ normal mode at time $t = 0$ (Figure 8.2).

Listing 8.3 Normal modes of a vibrating circular drum

```
# eg8-drum-normal-modes.py
import numpy as np
from scipy.special import jn, jn_zeros
import pylab

# Allow calculations up to m = mmax
mmax = 5

def displacement(n, m, r, theta):
    """
    Calculate the displacement of the drum membrane at (r, theta; t=0)
    in the normal mode described by integers n >= 0, 0 < m <= mmax.

    """

    # Pick off the mth zero of Bessel function Jn
    k = jn_zeros(n, mmax+1)[m]
    return np.sin(n*theta) * jn(n, r*k)

# Positions on the drum surface are specified in polar coordinates
r = np.linspace(0, 1, 100)
theta = np.linspace(0, 2 * np.pi, 100)

# Create arrays of cartesian coordinates (x, y) ...
x = np.array([rr*np.cos(theta) for rr in r])
y = np.array([rr*np.sin(theta) for rr in r])
# ... and vertical displacement (z) for the required normal mode at
# time, t = 0
```

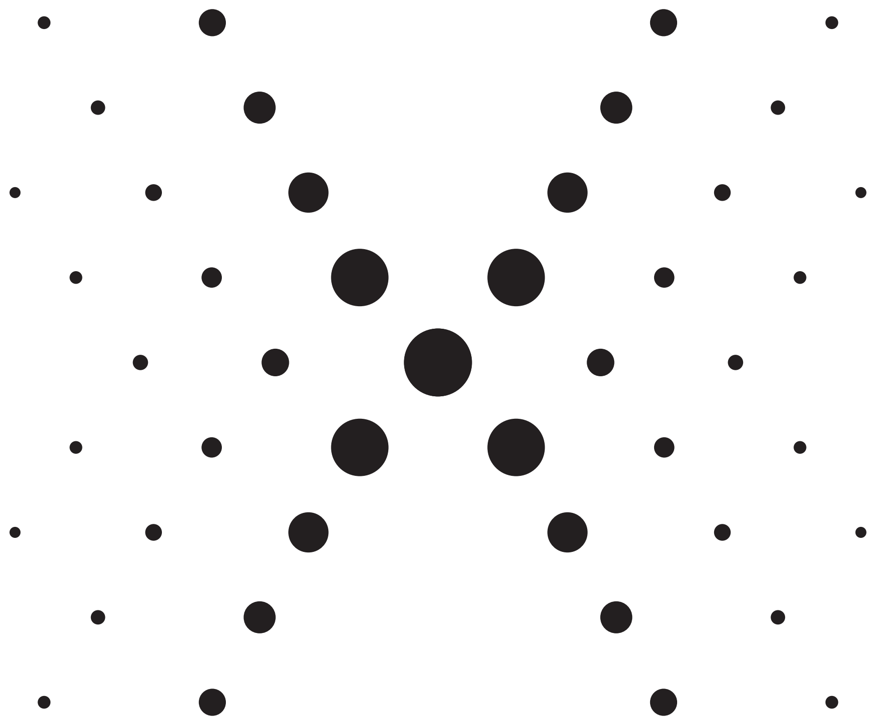


Figure 8.3 The diffraction pattern of a uniform, continuous helix.

```
n, m = 3, 2
z = np.array([displacement(n, m, rr, theta) for rr in r])

pylab.contour(x, y, z)
pylab.show()
```

Example E8.4 In an important paper in 1953.[4] Rosalind Franklin published the X-ray diffraction pattern of DNA from calf thymus, which displays a characteristic X shape of diffraction spots indicative of a helical structure.

The diffraction pattern of a uniform, continuous helix consists of a series of "layer lines" of spacing $1/p$ in reciprocal space where p is the helix pitch (the height of one complete turn of the helix, measured parallel to its axis). The intensity distribution along the nth layer line is proportional to the square of the nth Bessel function, $J_n(2\pi rR)$, where r is the radius of the helix and R is the radial coordinate in reciprocal space.

Consider the diffraction pattern of a helix with $p = 34$ Å and $r = 10$ Å. The code listing here produces an SVG image of the diffraction pattern of a helix (Figure 8.3).

[4] R. E. Franklin, R. G. Gosling, (1953). *Nature* **171**, 740.

Listing 8.4 Generating an image of the diffraction pattern of a uniform, continuous helix

```
# eg8-dna-diffraction.py

import numpy as np
from scipy.special import jn
import pylab

# Vertical range of the diffraction pattern: plot nlayer line layers above and
# below the center horizontal
nlayers = 5
ymin, ymax = -nlayers, nlayers

# Horizontal range of the diffraction pattern, x = 2pi.r.R
xmin, xmax = -10, 10
npts = 4000
x = np.linspace(xmin, xmax, npts)

# Diffraction pattern along each line layer: |Jn(x)|^2
# for n = 0, 1, ..., nlayers-1
❶ layers = np.array([jn(i, x)**2 for i in range(nlayers)])

# Obtain the indexes of the maxima in each layer
❷ maxi = [(np.diff(np.sign(np.diff(layers[i,:]))) < 0).nonzero()[0] + 1
                                                   for i in range(nlayers)]

# Create the SVG image, using circles of different radii for diffraction spots
svg_name='eg8-dna-diffraction.svg'
canvas_width = canvas_height = 500
fo = open(svg_name, 'w')
print("""<?xml version="1.0" encoding="utf-8"?>
         <svg xmlns="www.w3.org/2000/svg"
              xmlns:xlink="www.w3.org/1999/xlink"
              width="{}" height="{}" style="background: {}">""".format(
                canvas_width, canvas_height, '#ffffff'), file=fo)

def svg_circle(r, cx, cy):
    """ Return the SVG mark up for a circle of radius r centered at (cx,cy). """
    return r'<circle r="{}" cx="{}" cy="{}"/>'.format(r, cx, cy)

# For each spot in each layer, draw a circle on the canvas. The circle radius
# is the scaled value of the diffraction intensity maximum, with a ceiling
# value of spot_max_radius because the center spots are very intense
spot_scaling, spot_max_radius = 50, 20
for i in range(nlayers):
    for j in maxi[i]:
❸       sx = (x[j] - xmin)/(xmax-xmin) * canvas_width
        sy = (i - ymin)/(ymax - ymin) * canvas_height
        spot_radius = min(layers[i,j]*spot_scaling, spot_max_radius)
        print(svg_circle(spot_radius, sx, sy), file=fo)
        if i:
            # The pattern is symmetric about the center horizontal:
            # duplicate the layers with i > 0
            sy = canvas_height - sy
            print(svg_circle(spot_radius, sx, sy), file=fo)

print(r'</svg>', file=fo)
```

❶ The two-dimensional array, `layers`, holds the diffraction intensity in each line layer, calculated as the square of a Bessel function.

❷ For plotting the pattern, we need to find the indexes of the maxima in the `layers` array: this line of code finds these maxima by determining where the *differences* between neighboring items go from positive to negative.

❸ Map the `(x,y)` coordinates in the reciprocal space of the diffraction pattern onto the canvas coordinates, `(sx,sy)`.

8.1.3 The gamma and beta functions; elliptic integrals

The gamma function is defined by the improper integral

$$\Gamma(x) = \int_0^\infty t^{x-1}e^{-t}\,\mathrm{d}t,$$

for real $x > 0$, and extended to negative x and complex numbers by analytic continuation. It occurs frequently in integration problems, combinatorics and in expressions for other special functions.

The gamma function and its natural logarithm are returned by the functions `gamma(x)` and `gammaln(x)`. There are also methods for the evaluation of the incomplete gamma functions (obtained by replacing the lower or upper limits in the integral above with the parameter `a`) and their inverses; these will not be described in detail here.

Example E8.5 The gamma function is related to the factorial by $\Gamma(x) = (x-1)!$ and both are plotted in the code mentioned later (see Figure 8.4). Note that $\Gamma(x)$ is not defined for negative integer x, which leads to discontinuities in the plot.

Listing 8.5 The Gamma function on the real line

```
# eg3-gamma.py
import numpy as np
from scipy.special import gamma
import pylab

# The Gamma function
ax = pylab.linspace(-5, 5, 1000)
pylab.plot(ax, gamma(ax), ls='-', c='k', label='$\Gamma(x)$')

# (x-1)! for x = 1, 2, ..., 6
ax2 = pylab.linspace(1,6,6)
xm1fac = np.array([1, 1, 2, 6, 24, 120])
pylab.plot(ax2, xm1fac, marker='*', markersize=12, markeredgecolor='r',
           ls='',c='r', label='$(x-1)!$')

pylab.ylim(-50,50)
pylab.xlim(-5, 5)
pylab.xlabel('$x$')
pylab.legend()
pylab.show()
```

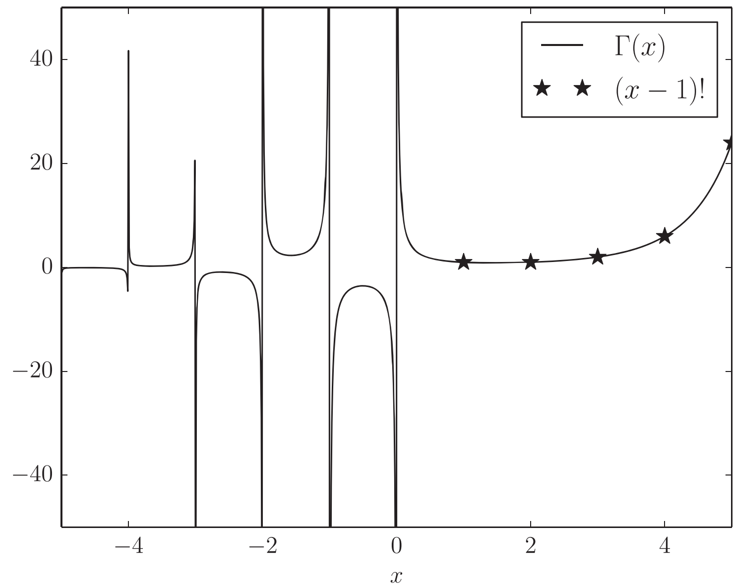


Figure 8.4 The gamma function on the real line, $\Gamma(x)$, and $(x-1)!$ for integer $x > 0$.

The beta function is defined by the definite integral

$$B(a,b) = \int_0^1 t^{a-1}(1-t)^{b-1}\,\mathrm{d}t, \quad a > 0, b > 0.$$

It is closely related to the gamma function: $B(a,b) = \Gamma(a)\Gamma(b)/\Gamma(a+b)$. The `scipy.special` functions `beta(a, b)` and `betaln(a,b)` return the beta function and its natural logarithm respectively. As with the gamma function, there is an *incomplete beta function*, $B(a,b;x)$, obtained by replacing the upper limit with x; the methods `betainc(a, b, x)` and `betaincinv(a, b, y)` return this function and its inverse.

Example E8.6 The exact classical mechanical description of a pendulum is quite complex, and the equations of motion usually only solved in introductory texts for small displacements about equilibrium. In this case, the period, $T \approx 2\pi\sqrt{L/g}$, and the motion is harmonic.

The general solution requires elliptic integrals, but the special case of a pendulum making 180° swings (i.e., ±90° about its equilibrium position) leads to the following expression for the period:

$$T = 2\sqrt{\frac{2l}{g}}\int_0^{\pi/2}\frac{\mathrm{d}\theta}{\sqrt{\cos\theta}}.$$

The substitution $x = \sin^2\theta$ transforms this integral into a beta function:

$$\int_0^{\pi/2}\frac{\mathrm{d}\theta}{\sqrt{\cos\theta}} = \tfrac{1}{2}\int_0^1 x^{-1/2}(1-x)^{-3/4}\,\mathrm{d}x = \tfrac{1}{2}B\left(\tfrac{1}{2},\tfrac{1}{4}\right).$$

Therefore,

$$T = \sqrt{2}B\left(\tfrac{1}{2}, \tfrac{1}{4}\right)\sqrt{\frac{l}{g}}.$$

To find the period of the pendulum in units of $\sqrt{l/g}$:

```
In [x]: import numpy as np
In [x]: from scipy.special import beta
In [x]: np.sqrt(2) * beta(0.5, 0.25)
7.4162987092054875
```

(Compare with the harmonic approximation, $2\pi = 6.283185$.)

The group of elliptic integrals and related functions form an important class of mathematical objects and have been widely studied. They find application in geometry, cryptography, analysis and many areas of physics. The complete elliptic integrals of the first and second kind, $K(m)$ and $E(m)$, are defined for $0 \le m \le 1$ by

$$K(m) = \int_0^{\pi/2} \frac{\mathrm{d}\theta}{\sqrt{1 - m\sin^2\theta}},$$
$$E(m) = \int_0^{\pi/2} \sqrt{1 - m\sin^2\theta}\;\mathrm{d}\theta.$$

Their values for the parameter m are returned by the functions `ellipk(m)` and `ellipe(m)`. The incomplete elliptic integrals (defined by replacing the upper limit of $\pi/2$ with the variable ϕ) are returned by `ellipkinc(phi, m)` and `ellipeinc(phi, m)` respectively:[5]

$$K(\phi, m) = \int_0^{\phi} \frac{\mathrm{d}\theta}{\sqrt{1 - m\sin^2\theta}},$$
$$E(\phi, m) = \int_0^{\phi} \sqrt{1 - m\sin^2\theta}\;\mathrm{d}\theta.$$

Example E8.7 The problem of finding an arc length of an ellipse is the origin of the name of the elliptic integrals. The equation of an ellipse with semi-major axis, a, and semi-minor axis, b, may be written in parametric form as

$$x = a\sin\phi$$
$$y = b\cos\phi$$

[5] It is necessary to be very careful with the notation of elliptic integrals; many sources use $F(\phi, m)$ instead of $K(\phi, m)$ for the first kind, define them with interchanged arguments (i.e., $F(m, \phi)$) or use the parameter k^2 instead m

$$F(\phi, k) = F(\phi|k^2) = \int_0^{\phi} \frac{\mathrm{d}\theta}{\sqrt{1 - k^2\sin^2\theta}}$$
$$E(\phi, k) = E(\phi|k^2) = \int_0^{\phi} \sqrt{1 - k^2\sin^2\theta}\,\mathrm{d}\theta.$$

The element of length along the ellipse's perimeter,

$$\begin{aligned} \mathrm{d}s &= \sqrt{\mathrm{d}x^2 + \mathrm{d}y^2} = \sqrt{a^2\cos^2\phi + b^2\sin^2\phi}\;\mathrm{d}\phi \\ &= a\sqrt{1 - e^2\sin^2\phi}\;\mathrm{d}\phi, \end{aligned}$$

where $e = \sqrt{1 - b^2/a^2}$ is the *eccentricity*. The arc length may therefore be written in terms of incomplete elliptic integrals of the second kind:

$$\int \mathrm{d}s = a\int_{\phi_1}^{\phi_2} \sqrt{1 - e^2\sin^2\phi}\;\mathrm{d}\phi = a[E(e;\phi_2) - E(e;\phi_1)].$$

Earth's orbit is an ellipse with semi-major axis 149,598,261 km and eccentricity 0.01671123. We will find the distance traveled by the Earth in one orbit, and compare it with that obtained assuming a circular orbit of radius 1 AU $\equiv$ 149597870.7 km.

The perimeter of an ellipse may be written using the earlier expression with $\phi_1 = 0, \phi_2 = 2\pi$:

$$P = a[E(e, 2\pi) - E(e, 0)] = 4aE(e),$$

since the entire perimeter is four times the quarter-perimeters, which may be written in terms of the *complete* elliptic integral of the second kind. We have

```
In [x]: import numpy as np
In [x]: from scipy.special import ellipe
In [x]: a, e  = 149598261, 0.01671123   # semi-major axis (km), eccentricity
In [x]: pe = 4 * a * ellipe(e)
In [x]: print(pe)
936014259.33                  # "exact" answer
In [x]: AU = 149597870.7      # mean orbit radius, km
In [x]: pc = 2 * np.pi * AU
In [x]: print(pc)
939951143.1675915             # assuming circular orbit
In [x]: (pc - pe) / pe * 100
0.42060084000247772
```

That is, the percentage error in the perimeter in treating the orbit as circular is about 0.42%.

8.1.4 The error function and related integrals

The error function, defined by:

$$\mathrm{erf}(z) = \frac{2}{\sqrt{\pi}}\int_0^z e^{-t^2}\,\mathrm{d}t$$

for real or complex z does not have a simple closed-form expression and so must be calculated numerically. `scipy.special` has several functions relating to the error function:

- `erf(z)`: the error function;
- `erfc(z)`: the complementary error function, $\mathrm{erfc}(z) = 1 - \mathrm{erf}(z)$. It is more accurate to use this function for large z than directly subtracting $\mathrm{erf}(z)$ from 1;

- `erfcx(z)`: the *scaled complementary error function*, $e^{z^2}\mathrm{erfc}(z)$;
- `erfinv(y)`: the inverse error function;
- `erfcinv(y)`: the inverse complementary error function;
- `wofz(z)`: the Faddeeva function, a scaled complementary error function with complex argument:

$$w(z) = e^{-z^2}\mathrm{erfc}(-iz) = \mathrm{erfcx}(-iz),$$

which appears in problems related to plasma physics and radiative transfer;
- `dawsn(z)`: the related integral known as Dawson's integral:

$$D(z) = e^{-z^2}\int_0^z e^{t^2}\,\mathrm{d}t.$$

Example E8.8 The wavefunction corresponding to the ground state of the one-dimensional quantum harmonic oscillator may be written as follows in terms of a parameter $\alpha = \sqrt{mk}/\hbar$, where m is the mass and k the oscillator force constant.

$$\psi_0(x) = \left(\frac{\alpha}{\pi}\right)^{1/4}\exp\left(-\alpha x^2/2\right)$$

The probability density of the oscillator's position is given by $P_0(x) = |\psi_0(x)|^2$ and is nonzero outside the classical turning points, $\pm\alpha^{-1/2}$, a phenomenon known as tunneling. We will calculate the probability of tunneling for an oscillator in the state ψ_0.

The wavefunction is symmetric about $x = 0$, so the probability of tunneling is

$$P(x < -\alpha) + P(x > \alpha) = 2P(x > \alpha) = 2\sqrt{\frac{\alpha}{\pi}}\int_{\alpha^{-1/2}}^{\infty}\exp\left(-\alpha x^2\right)\,\mathrm{d}x$$
$$= \frac{2}{\sqrt{\pi}}\int_1^{\infty}e^{-y^2}\,\mathrm{d}y = \mathrm{erfc}(1).$$

The complementary error function can be calculated directly:

```
In [x]: from scipy.special import erfc
In [x]: erfc(1)
0.15729920705028516
```

or about 16%.

Example E8.9 The *Voigt line profile* occurs in the modeling and analysis of radiative transfer in the atmosphere. It is the convolution of a Gaussian profile, $G(x;\sigma)$, and a Lorentzian profile, $L(x;\gamma)$:

$$V(x;\sigma,\gamma) = \int_{-\infty}^{\infty} G(x';\sigma)L(x-x';\gamma)\,\mathrm{d}x' \quad \text{where}$$

$$G(x;\sigma) = \frac{1}{\sigma\sqrt{2\pi}}\exp\left(\frac{-x^2}{2\sigma}\right) \quad \text{and} \quad L(x;\gamma) = \frac{\gamma/\pi}{x^2+\gamma^2}.$$

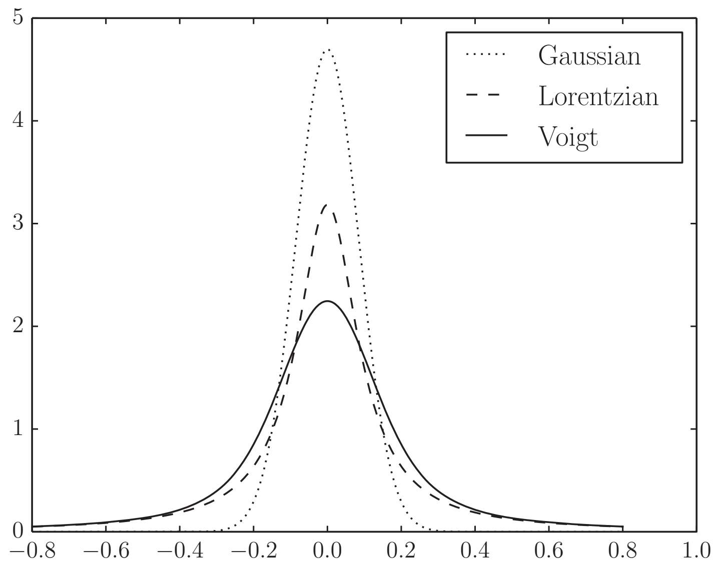


Figure 8.5 A comparison of the Lorentzian, Gaussian and Voigt line shapes with $\gamma = \alpha = 0.1$.

Here γ is the half-width at half-maximum (HWHM) of the Lorentzian profile and σ is the standard deviation of the Gaussian profile, related to its HWHM, α, by $\alpha = \sigma\sqrt{2\ln 2}$. In terms of frequency, ν, $x = \nu - \nu_0$ where ν_0 is the line center.

There is no closed form for the Voigt profile, but it is related to the real part of the Faddeeva function, $w(z)$ by

$$V(x;\sigma,\gamma) = \frac{\mathrm{Re}\,[w(z)]}{\sigma\sqrt{2\pi}}, \text{ where } z = \frac{x+i\gamma}{\sigma\sqrt{2}}.$$

The program mentioned here plots the Voigt profile for $\gamma = 0.1, \alpha = 0.1$ and compares it with the corresponding Gaussian and Lorentzian profiles (Figure 8.5). The equations mentioned earlier are implemented in the three functions, `G`, `L` and `V`, defined in the code here.

Listing 8.6 A comparison of the Lorentzian, Gaussian and Voigt line shapes

```
# eg8-voigt.py
import numpy as np
from scipy.special import wofz
import pylab

def G(x, alpha):
    """ Return Gaussian line shape at x with HWHM alpha """
    return np.sqrt(np.log(2) / np.pi) / alpha\
                             * np.exp(-(x / alpha)**2 * np.log(2))

def L(x, gamma):
    """ Return Lorentzian line shape at x with HWHM gamma """
    return gamma / np.pi / (x**2 + gamma**2)

def V(x, alpha, gamma):
    """
    Return the Voigt line shape at x with Lorentzian component HWHM gamma
    and Gaussian component HWHM alpha.
```

```
    """
    sigma = alpha / np.sqrt(2 * np.log(2))

    return np.real(wofz((x + 1j*gamma)/sigma/np.sqrt(2))) / sigma\
                                                         / np.sqrt(2*np.pi)

alpha, gamma = 0.1, 0.1
x = np.linspace(-0.8,0.8,1000)
pylab.plot(x, G(x, alpha), ls=':', c='k', label='Gaussian')
pylab.plot(x, L(x, gamma), ls='--', c='k', label='Lorentzian')
pylab.plot(x, V(x, alpha, gamma), c='k', label='Voigt')
pylab.legend()
pylab.show()
```

8.1.5 Fresnel integrals

The Fresnel integrals are encountered in optics and are defined by the equations

$$S(z) = \int_0^z \sin\left(\frac{\pi t^2}{2}\right)\,\mathrm{d}t, C(z) = \int_0^z \cos\left(\frac{\pi t^2}{2}\right)\,\mathrm{d}t.$$

Both are returned in a tuple for real or complex argument `z` by the `special.scipy` function `fresnel(z)`. The related function, `fresnel_zeros(nt)`, returns the first `nt` complex zeros of $S(z)$ and $C(z)$.

Example E8.10 As well as playing an important role in the description of diffraction effects in optics, the Fresnel integrals find an application in the design of motorway junctions (freeway intersections). The curve described by the parametric equations $(x, y) = (S(t), C(t))$ is called a *clothoid* (or Euler spiral) and has the property that its curvature is proportional to the distance along the path of the curve. Hence, a vehicle traveling at constant speed will experience a constant rate of angular acceleration as it travels around the curve – this means that the driver can turn the steering wheel at a constant rate, which makes the junction safer.

The following code plots the Euler spiral for $-10 \leq t \leq 10$ (Figure 8.6).

```
In [x]: import numpy as np
In [x]: from scipy.special import fresnel
In [x]: import pylab
In [x]: t = np.linspace(-10, 10, 1000)
In [x]: pylab.plot(*fresnel(t), c='k')
In [x]: pylab.show()
```

8.1.6 Binomial coefficients and exponential integrals

The binomial coefficient $\binom{n}{k} = {}^nC_k$ is returned by the `scipy.special` function `binom(n,k)`.

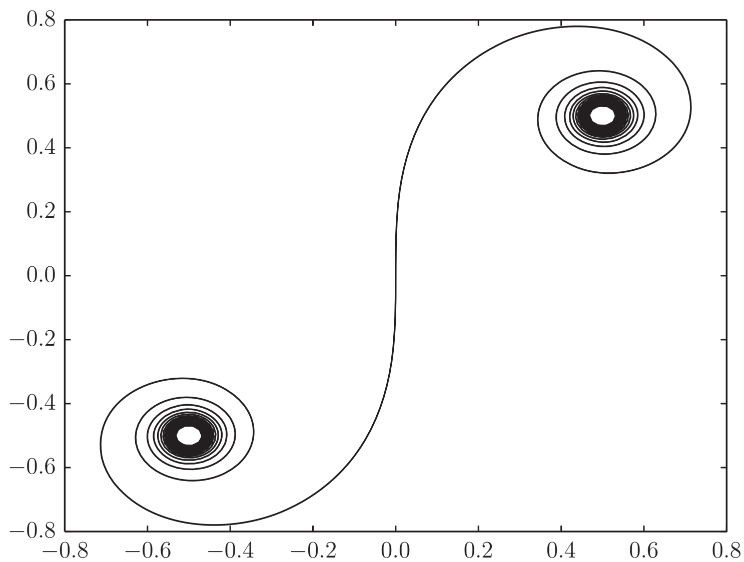


Figure 8.6 The Euler spiral.

Various functions are supplied for the evaluation of different forms of the exponential integral. The standard form is returned by `expi(z)`:

$$\mathrm{Ei}(z) = \int_{-\infty}^{z} \frac{e^t}{t}\,\mathrm{d}t, \quad |\arg(-z) < \pi|$$

`expn(n, x)` returns the value of

$$\int_1^\infty \frac{e^{-xt}}{t^n}\,\mathrm{d}t.$$

For $n = 1$, it is faster and more accurate to use `exp1(z)`:

$$\int_1^\infty \frac{e^{-zt}}{t}\,\mathrm{d}t.$$

Example E8.11 Any integral of the form

$$\int f(z)e^z\,\mathrm{d}z,$$

where $f(z) = P(z)/Q(z)$ is a rational function, can reduced to the form

$$\int R(z)e^z\,\mathrm{d}z + \sum_i \int \frac{e^z}{(z-a_i)^{n_i}}\,\mathrm{d}z,$$

where $R(z)$ is a polynomial (which may be zero) by expansion in partial fractions. The first integral here can be evaluated by standard methods (repeated integration by parts). Provided the path of integration does not pass through any singular points of the integrand, the second term can be written in terms of exponential integrals.

For example, consider the integral

$$I = \int_{-\infty}^{-2} \frac{e^z}{z^2(z-1)}\,\mathrm{d}z.$$

It can easily be shown that

$$\frac{1}{z^2(z-1)} = \frac{1}{z-1} - \frac{1}{z} - \frac{1}{z^2}$$

and so the integral may be written as the three terms

$$I = \int_{-\infty}^{-2} \frac{e^z}{z-1}\,\mathrm{d}z - \int_{-\infty}^{-2} \frac{e^z}{z}\,\mathrm{d}z - \int_{-\infty}^{-2} \frac{e^z}{z^2}\,\mathrm{d}z.$$

The second integral is simply $-\mathrm{Ei}(-2)$ and substitution $u = z - 1$ resolves the first integral to $e\mathrm{Ei}(-3)$. The last integral may be written in terms of $\mathrm{En}(z)$ or further reduced by integration by parts to

$$\int_{-\infty}^{-2} \frac{e^z}{z^2}\,\mathrm{d}z = -\frac{e^{-2}}{2} + \mathrm{Ei}(-2).$$

Therefore,

$$I = e\mathrm{Ei}(-3) - 2\mathrm{Ei}(-2) - \frac{e^{-2}}{2}.$$

In SciPy,

```
In [x]: import numpy as np
In [x]: from scipy.special import expi
In [x]: np.e * expi(-3) - 2*expi(-2) - np.exp(-2)/2
-0.0053357974213484663
```

8.1.7 Orthogonal polynomials and spherical harmonics

There are a large number of functions in `scipy.special` for the evaluation of different sorts of orthogonal polynomials, including the Legendre, Jacobi, Laguerre, Hermite and different flavors of Chebyshev polynomials. They take the general name `eval_poly(n, x)` where n is the order of the polynomial and `x` is an array-like sequence of values at which to evaluate the polynomial. Table 8.2 gives the names of some of these functions.

The spherical harmonics used in SciPy are defined by the formula

$$Y_n^m(\phi, \theta) = \sqrt{\frac{(2n+1)}{4\pi}\frac{(n-m)!}{(n+m)!}} P_n^m(\cos\phi)e^{im\theta},$$

where $n = 0, 1, 2, \cdots$ is called the *degree* and $m = -n, -n+1, \cdots n$ the *order* of the spherical harmonic. The functions $P_n^m(x)$ are the associated Legendre polynomials. As with so many special functions, different fields adopt different phase conventions and normalizations, so it is important to check these carefully and make the appropriate

Table 8.2 Some of the orthogonal polynomials in SciPy

Function	Description
`eval_legendre(n, x)`	Legendre polynomial, $P_n(x)$
`eval_chebyt(n, x)`	Chebyshev polynomial of the first kind, $T_n(x)$
`eval_chebyu(n, x)`	Chebyshev polynomial of the second kind, $U_n(x)$
`eval_hermite(n, x)`	(Physicists') Hermite polynomial, $H_n(x)$
`eval_jacobi(n, alpha, beta, x)`	Jacobi polynomial, $P_n^{(\alpha,\beta)}(x)$
`eval_laguerre(n, x)`	Laguerre polynomial of the first kind, $L_n(x)$
`eval_genlaguerre(n, alpha x)`	Generalized Laguerre polynomial of the first kind, $L_n^{\alpha}(x)$

modifications when using them. In particular, many other fields use l for the degree of the harmonic and reverse the definition of θ and ϕ. To be clear, in SciPy θ is the *azimuthal* (longitudinal) angle (taking values between 0 and 2π) and ϕ is the polar (colatitudinal) angle (between 0 and π).

The `scipy.special.sph_harm` method is called with the arguments:

```
scipy.special.sph_harm(m, n, theta, phi)
```

where `theta` and `phi` can be array-like objects.

Example E8.12 Visualizing the spherical harmonics is a little tricky because they are complex and defined in terms of angular coordinates, (θ, ϕ). One way is to plot the real part only on the unit sphere. Matplotlib provides a toolkit for such 3D plots, `mplot3d`, as illustrated by the following code which produces Figure 8.7.[6]

Listing 8.7 The spherical harmonic defined by $l = 3, m = 2$

```
# eg8-spherical-harmonics.py

import matplotlib.pyplot as plt
from matplotlib import cm, colors
from mpl_toolkits.mplot3d import Axes3D
import numpy as np
from scipy.special import sph_harm

phi = np.linspace(0, np.pi, 100)
theta = np.linspace(0, 2*np.pi, 100)
phi, theta = np.meshgrid(phi, theta)

# The Cartesian coordinates of the unit sphere
x = np.sin(phi) * np.cos(theta)
y = np.sin(phi) * np.sin(theta)
z = np.cos(phi)

m, l = 2, 3
```

[6] See Section 7.2.3 and http://matplotlib.org/mpl_toolkits/mplot3d/.

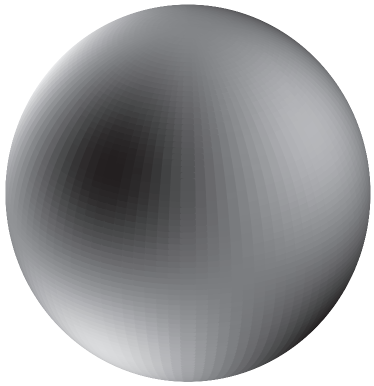


Figure 8.7 A depiction of the spherical harmonic defined by $l = 3, m = 2$.

```
# Calculate the spherical harmonic Y(l,m) and normalize to [0,1]
fcolors = sph_harm(m, l, theta, phi).real
fmax, fmin = fcolors.max(), fcolors.min()
fcolors = (fcolors - fmin)/(fmax - fmin)

# Set the aspect ratio to 1 so our sphere looks spherical
fig = plt.figure(figsize=plt.figaspect(1.))
ax = fig.add_subplot(111, projection='3d')
ax.plot_surface(x, y, z,  rstride=1, cstride=1, facecolors=cm.jet(fcolors))
# Turn off the axis planes
ax.set_axis_off()
plt.show()
```

8.1.8 Exercises

Questions

Q8.1.1 By changing a single line in the program of Example E8.1, output the 10 *most accurately* known constants (excluding those set to their values by definition).

Q8.1.2 Use SciPy's constants and conversion factors to calculate the number density, N/V, of ideal gas molecules at standard temperature and pressure ($T = 0$ °C, $p = 1$ atm). The ideal gas law is $pV = Nk_\mathrm{B}T$.

Problems

P8.1.1 Use `scipy.special.binom` to create a depiction of Pascal's triangle of binomial coefficients $\binom{n}{k}$ up to $n = 8$.

P8.1.2 The *Airy pattern* is the circular diffraction pattern of resulting from a uniformly illuminated circular aperture. It consists of a bright, central disc surrounded by fainter rings. Its mathematical description may be written in terms of the Bessel function of the

first kind,

$$I(\theta) = I_0 \left(\frac{2J_1(x)}{x} \right)^2,$$

where θ is the observation angle and $x = ka \sin\theta$. a is the aperture radius and $k = 2\pi/\lambda$ is the *angular wavenumber* of the light with wavelength λ.

Plot the Airy pattern as $I(x)/I_0$ for $-10 \le x \le 10$ and deduce from the position of the first minimum in this function the maximum resolving power (in arcsec) of the human eye (pupil diameter 3 mm) at a wavelength of 500 nm.

P8.1.3 Write a function, `get_wv`, which takes a molar bond dissociation energy, `D0`, in $\mathrm{kJ\,mol^{-1}}$ and returns the wavelength of a photon corresponding to that energy *per molecule*, in nm. The energy of a photon with wavelength λ is $E = hc/\lambda$.

For example,

```
In [x]: get_wv(497)
Out[x}: 240.69731528286377
```

P8.1.4 An *ellipsoid* is the three-dimensional figure bounded by the surface described by the equation

$$\frac{x^2}{a^2} + \frac{y^2}{b^2} + \frac{z^2}{c^2} = 1,$$

where a, b and c are the *semi-principal axes*. If $a = b = c$, the ellipsoid is a sphere. The volume of an ellipsoid has a simple form,

$$V = \frac{4}{3}\pi abc.$$

There is no closed formula for the surface area of a general ellipsoid, but it may be expressed in terms of incomplete elliptic integrals of the first and second kinds, $K(\phi, k)$ and $E(\phi, k)$:

$$S = 2\pi c^2 + \frac{2\pi ab}{\sin\phi}\left(K(\phi, k^2)\cos^2\phi + E(\phi, k^2)\sin^2\phi \right),$$

where

$$\cos\phi = \frac{c}{a}, \quad k = \frac{a\sqrt{b^2 - c^2}}{b\sqrt{a^2 - c^2}}$$

and the coordinate system has been chosen such that $a \ge b \ge c$.

Define a function, `ellipsoid_surface`, to calculate the surface area of a general ellipsoid, and compare the results for different-shaped ellipsoids with the following approximate formula:

$$S \approx 2\pi c^2 + 2\pi abr\left(1 - \frac{b^2 - c^2}{6b^2}r^2\left(1 - \frac{3b^2 + 10c^2}{56b^2}r^2\right)\right),$$

$$\text{where} \quad r = \frac{\phi}{\sin\phi}.$$

P8.1.5 The *drawdown* or change in hydraulic head, s (a measure of the water pressure above some geodetic datum), a distance r from a well at time t, from which water is being pumped at a constant rate, Q, can be modeled using the *Theis* equation,

$$s(r,t) = H_0 - H(r,t) = \frac{Q}{2\pi T}W(u), \quad \text{where} \quad u = \frac{r^2 S}{4Tt}.$$

Here H_0 is the hydraulic head in the absence of the well, S is the aquifer storage coefficient (volume of water released per unit decrease in H per unit area) and T is the transmissivity (a measure of how much water is transported horizontally per unit time). The *Well Function*, $W(u)$ is simply the exponential integral, $E_1(u)$.

For a well being pumped at $Q = 1,000\ \mathrm{m^3\,day^{-1}}$ from an aquifer described by the parameters $H_0 = 20$ m, $S = 0.0003$, $T = 1,000\ \mathrm{m^2\,day^{-1}}$, determine the height of the hydraulic head as a function of r after $t = 1$ day of pumping.

Compare your answer with the approximate version of the Theis equation known as the Jacob equation, in which the well function is taken to be appoximately $W(u) \approx -\gamma - \ln u$ where $\gamma = 0.577215664\cdots$ is the Euler-Mascheroni constant.

P8.1.6 Some electronic components are cooled by annular fins (heatsinks) which conduct heat away from the component and provide a larger surface area for that heat to dissipate to the surroundings.

The cooling efficiency of an annular fin of width $2w$ and inner and outer radii r_0 and r_1 may be written in terms of modified Bessel functions of the first and second kinds:

$$\eta = \frac{2r_0}{\beta(r_1^2 - r_0^2)} \frac{K_1(u_0)I_1(u_1) - I_1(u_0)K_1(u_1)}{K_0(u_0)I_1(u_1) + I_0(u_0)K_1(u_1)},$$

where $u_0 = \beta r_0$, $u_1 = \beta r_1$ and

$$\beta = \sqrt{\frac{h_c}{\kappa w}}.$$

h_c is the heat transfer coefficient (which is taken to be constant over the fin's surface) and κ is the thermal conductivity of the fin material.

What is the cooling efficiency of an aluminium annular fin with dimensions $r_0 = 5$ mm, $r_1 = 10$ mm, $w = 0.1$ mm? Take $h_c = 10\ \mathrm{W\,m^{-2}\,K^{-1}}$ and $\kappa = 200\ \mathrm{W\,m^{-1}\,K^{-1}}$.

Calculate the heat dissipation, $\dot{Q}$ (the product of the efficiency, the fin area and the temperature difference) for a component temperature of $T_0 = 400$ K and ambient temperature $T_\mathrm{e} = 300$ K.

8.2 Integration and ordinary differential equations

The `scipy.integrate` package contains functions for computing definite integrals. It can evaluate both proper (with finite limits) and improper (infinite limits) integrals. It can also perform numerical integration of systems of ordinary differential equations.

8.2.1 Definite integrals of a single variable

The basic numerical integration routine is `scipy.integrate.quad`, which is based on the venerable FORTRAN 77 QUADPACK library. It uses adaptive quadrature to approximate the value of an integral by dividing its domain into subintervals that are chosen iteratively to meet a particular tolerance (that is, estimated absolute or relative error). In its simplest form, it takes three arguments: a Python function object corresponding to the function to integrate, `func`, and the limits of integration, `a` and `b`. `func` must take at least one argument; if it takes more than one it is integrated along the coordinate corresponding to the first argument. In simple usage, `lambda` expressions are a convenient way to define `func`. For example, to evaluate $\int_1^4 x^{-2}\,\mathrm{d}x = \frac{3}{4}$ numerically:

```
In [x]: from scipy.integrate import quad
In [x]: f = lambda x: 1/x**2
Out[x]: quad(f, 1, 4)
(0.7500000000000002, 1.913234548258995e-09)
```

`quad` returns two values in a tuple – the value of the integral and an estimate of the absolute error in the result.

Use `np.inf` to evaluate improper integrals:

```
In [x]: quad(lambda x: np.exp(-x**2), 0, np.inf)
Out[x]: (0.8862269254527579, 7.101318390472462e-09)
In [x]: np.sqrt(np.pi)/2    # analytical result
Out[x]: 0.88622692545275794
```

Note that in this call to `quad` we didn't even give the function a name but simply passed it as an anonymous `lambda` object.

More complicated functions require a Python function object defined with `def`:

```
In [x]: def g(x):
   ...:     if abs(x) < 0.5:
   ...:         return -x
   ...:     return x - np.sign(x)
   ...:
In [x]: quad(g, -0.6, 0.8)
Out[x]: (-0.06000000000000002, 6.661338147750941e-17)
```

Functions with singularities or discontinuities can cause problems for the numerical quadrature routine even if the required integral is well-defined. For example, the sinc function, $f(x) = \sin(x)/x$ has a removable singularity at $x = 0$, which causes the following simple application of `quad` to fail:

```
In [x]: sinc = lambda x: np.sin(x)/x
In [x]: quad(sinc, -2, 2)
...: RuntimeWarning: invalid value encountered in double_scalars
Out[37]: (nan, nan)
```

The solution is to configure `quad` by passing a list of such *break points* to the `points` argument (the list does not have to be ordered):

```
In [x] quad(sinc, -2, 2, points=[0,])
(3.210825953605389, 3.5647329017567276e-14)
```

Note that break points cannot be specified with infinite limits.

The arguments `epsrel` and `epsabs` allow the specification of a desired accuracy of the quadrature as a relative or absolute tolerance. The default values are both `1.49e-8`, but the integration can be done faster if a less-accurate answer is required. As an example, consider integrating the rapidly varying function, $f(x) = e^{-|x|}\sin^2 x^2$:

```
In [x]: f = lambda x: np.sin(x**2)**2 * np.exp(-np.abs(x))
In [x]: quad(f, -1, 2, epsabs=0.1)
Out[x]: (0.29551455828969975, 0.001529571827911671)
In [x]: quad(f, -1, 2, epsabs=1.49e-8)    # (the default absolute tolerance)
Out[x]: (0.29551455505239044, 4.449763315720537e-10)
```

Note that `epsabs` is only a requested upper bound: the actual estimated accuracy in the result may be much better, and in fact the actual result may be more accurate than this estimate.

If a function takes one or more parameters in addition to its principal argument, these need to be passed to `quad` as a tuple in `args`. For example, the integral

$$I_{n,m} = \int_{-\pi/2}^{\pi/2} \sin^n x \cos^m x \, \mathrm{d}x$$

can be evaluated numerically with

```
In [x]: def f(x, n, m):
   ....:     return np.sin(x)**n * np.cos(x)**m
   ....:
In [x]: n, m = 2, 1
In [x]: quad(f, -np.pi/2, np.pi/2, args=(n, m))
(0.6666666666666666, 1.625746841018571e-13)
```

Note that the additional parameters, `n` and `m` here, appear as arguments to our function *after* the coordinate to be integrated over (`x`).

Example E8.13 Consider a torus of average radius R and cross-sectional radius r. The volume of this shape may be evaluated analytically in Cartesian coordinates as a volume of revolution:

$$V = 2\int_{R-r}^{R+r} 2\pi xz \, \mathrm{d}x, \quad \text{where } z = \sqrt{r^2 - (x-R)^2}.$$

The center of the torus is at the origin and the z axis is taken to be its symmetry axis.

The integral is tedious but yields to standard methods: $V = 2\pi^2 R r^2$. Here we take a numerical approach with the values $R = 4$, $r = 1$:

```
In [x]: R, r = 4, 1
In [x]: f = lambda x, R, r: x * np.sqrt(r**2 - (x-R)**2)
In [x]: V, _ = quad(f, R-r, R+r, args=(R, r))
In [x]: V *= 4 * np.pi
In [x]: Vexact = 2 * np.pi**2 * R * r**2
In [x]: print('V = {} (exact: {})'.format(V, Vexact))
Out[x]: V = 78.95683520871499 (exact: 78.95683520871486)
```

8.2.2 Integrals of two and more variables

The `scipy.integrate` functions `dblquad`, `tplquad` and `nquad` evaluate double, triple and multiple integrals respectively. Because, in general, the limits on one coordinate may depend on another coordinate, the syntax for calling these functions is a little more complicated.

`dblquad` evaluates the double integral:

$$\int_a^b \int_{g(x)}^{h(x)} f(x, y)\, \mathrm{d}y\, \mathrm{d}x.$$

It is passed $f(x, y)$ as a function of at least two variables, `func(y, x, ...)`. The function must take `y` as its first argument and `x` as its second argument. The integral limits are passed to `dblquad` in four further arguments. First, the two arguments, `a` and `b`, specify the lower and upper limits on the x-integral respectively, as for `quad`. The next two arguments, `gfun` and `hfun`, are the lower and upper limits on the y-integral and they must be *callable objects* taking a single floating point argument, the value of `x` at which the limit applies (i.e., they must themselves be functions of `x`). If either of the y-integral limits does not depend on x, `gfun` or `hfun` can return a constant value.

As a simple example, the integral

$$\int_1^4 \int_0^2 x^2 y\, \mathrm{d}y \mathrm{d}x$$

can be evaluated with

```
In [x]: f = lambda y, x: x**2 * y
In [x]: a, b = 1, 4
In [x]: gfun = lambda x: 0
In [x]: hfun = lambda x: 2
In [x]: dblquad(f, a, b, gfun, hfun)
Out[x]: (42.00000000000001, 4.662936703425658e-13)
```

Here, `gfun` and `hfun` are each called with a value of `x`, but they return a constant (`0` and `2` respectively) no matter what this value is.

Of course, it is possible to wrap all of this into a single line:

```
In [x]: dblquad(lambda y, x: x**2 * y, 1, 4, lambda x: 0, lambda x: 2)
Out[x]: (42.00000000000001, 4.662936703425658e-13)
```

A double integral can be used to find the area of some two-dimensional shape bounded by one or more functions. For an example in polar coordinates, consider the area inside the curve $r = 2 + 2\sin\theta$ but outside the circle defined by $r = 2$ for θ in $[0, 2\pi]$ (see Figure 8.8). These curves intersect at $\theta = 0, \pi$ so the required integral is

$$A = \int_0^{\pi} \int_2^{2+2\sin\theta} r\, \mathrm{d}r\, \mathrm{d}\theta,$$

where $r\,\mathrm{d}r\,\mathrm{d}\theta$ is the infinitesimal area element in polar coordinates. This particular integral is fairly straightforward to evaluate analytically ($A = 8 + \pi$), so the numerical result is easy to check:

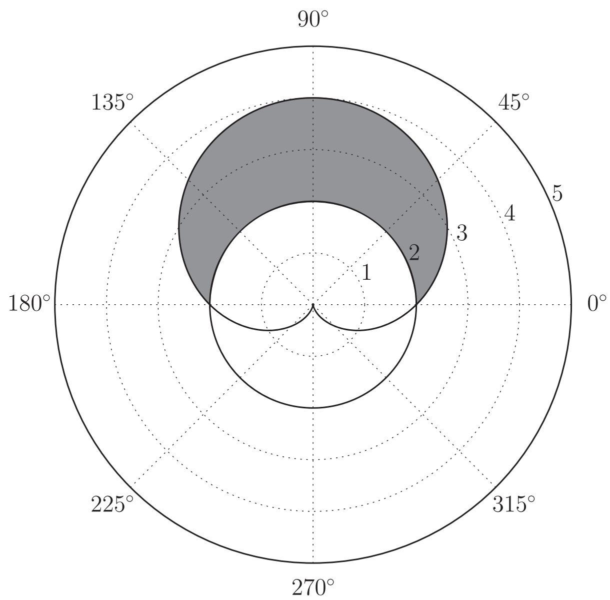


Figure 8.8 The region defined as the area inside $r = 2 + 2\sin\theta$ but outside the circle $r = 2$.

```
In [x]: r1, r2 = lambda theta: 2, lambda theta: 2 + 2*np.sin(theta)
In [x]: A, _ = dblquad(lambda r, theta: r, 0, np.pi, r1, r2)
Out[x]: 11.141592653589791
In [x]: 8 + np.pi        # exact answer
Out[x]: 11.141592653589793
```

The function to evaluate is simply `r`, defined by `lambda r, theta: r`; in the inner integral the limits on r are 2 and $2 + 2\sin\theta$; for the outer integral θ ranges from 0 to π.

The method `tplquad` evaluates triple integrals and takes a function of three variables, `func(z, y, x)` and *six* further arguments: constant x-limits, `a` and `b`, y-limits `gfun(x)` and `hfun(x)` (which are functions, as for `dblquad`, and z-limits `qfun(x, y)` and `rfun(x,y)` (functions of `x` and `y` in that order).

Higher dimensional integrations are handled by the `scipy.integrate.nquad` method which will not be discussed here (documentation and examples are available online).[7]

Example E8.14 The volume of the unit sphere, $4\pi/3$, can be expressed as a triple integral in spherical polar coordinates with constant limits:

$$\int_0^{2\pi}\int_0^{\pi}\int_0^{1} r^2 \sin\theta \; \mathrm{d}r\mathrm{d}\theta\mathrm{d}\phi.$$

```
In [x]: from scipy.integrate import tplquad
In [x]: tplquad(lambda phi, theta, r: r**2 * np.sin(theta),
                0, 1,
                lambda theta: 0, lambda theta: np.pi,
```

[7] http://docs.scipy.org/doc/scipy/reference/generated/scipy.integrate.nquad.html/scipy.integrate.nquad.

```
                lambda theta, phi: 0, lambda theta, phi: 2*np.pi)
Out[x]: (4.18879020478639, 4.650491330678174e-14)
```

Or in Cartesian coordinates with limits as functions:

$$8\int_0^1\int_0^{\sqrt{1-x^2}}\int_0^{\sqrt{1-x^2-y^2}}\mathrm{d}z\,\mathrm{d}y\,\mathrm{d}x,$$

where the integral is in the positive octant of the three-dimensonal Cartesian axes.

```
In [x]: A, _ = tplquad(lambda z, y, x: 1,
                        0, 1,
                        lambda x: 0, lambda x: np.sqrt(1 - x**2),
                        lambda x,y: 0, lambda x,y: np.sqrt(1 - x**2 - y**2))
In [x]: 8*A
Out[x]: 4.188790204786391
```

Example E8.15 This example finds the mass and center of mass of the tetrahedron bounded by the coordinate axes and the plane $x+y+z=1$ with density $\rho=\rho(x,y,z)$ where $\rho(x,y,z)$ is provided as a `lambda` function. We test it with the functions $\rho=1$, $\rho=x$ and $\rho=x^2+y^2+z^2$.

The mass may be written as a triple integral of the density over the volume of the tetrahedron:

$$m=\int_V\rho(x,y,z)\;\mathrm{d}V=\int_0^1\int_0^{1-x}\int_0^{1-x-y}\rho(x,y,z)\;\mathrm{d}z\,\mathrm{d}y\,\mathrm{d}x,$$

and the coordinates of the center of mass are given by

$$m\bar{x}=\int_V x\rho(x,y,z)\;\mathrm{d}V,\quad m\bar{y}=\int_V y\rho(x,y,z)\;\mathrm{d}V,\quad m\bar{z}=\int_V z\rho(x,y,z)\;\mathrm{d}V.$$

The following program uses `scipy.integrate.tplquad` to perform the necessary integrations (which can also be solved analytically).

Listing 8.8 Calculating the mass and center of mass of a tetrahedron given three different densities

```
# eg8-tetrahedron-cofm.py

import numpy as np
from scipy.integrate import tplquad

# The integration limits on x, y, z:
a, b = 0, 1
gfun, hfun = lambda x: 0, lambda x: 1 - x
qfun, rfun = lambda x, y: 0, lambda x, y: 1 - x - y
lims = (a, b, gfun, hfun, qfun, rfun)    ❶

# The three different density functions
rhos = [lambda x, y, z: 1,
        lambda x, y, z: x,
        lambda x, y, z: x**2 + y**2 + z**2]
```

```
for rho in rhos:
    # The mass as a triple integral of rho over the volume
    m, _ = tplquad(rho, *lims)
    # The center of mass (xbar, ybar, zbar)
    mxbar, _ = tplquad(lambda x, y, z: x * rho(x,y,z), *lims)
    mybar, _ = tplquad(lambda x, y, z: y * rho(x,y,z), *lims)
    mzbar, _ = tplquad(lambda x, y, z: z * rho(x,y,z), *lims)
    xbar, ybar, zbar = mxbar / m, mybar / m, mzbar / m

    print('mass = {:g}, CofM = ({:g}, {:g}, {:g})'.format(m, xbar, ybar, zbar))
```

❶ Note that the six arguments representing the limits on the triple integral (two constants and two pairs of `lambda` functions) have been packed into a tuple, `lims` (the parentheses are optional here).

The output is:

```
mass = 0.166667, CofM = (0.25, 0.25, 0.25)
mass = 0.0416667, CofM = (0.4, 0.2, 0.2)
mass = 0.05, CofM = (0.277778, 0.277778, 0.277778)
```

8.2.3 Ordinary differential equations

Ordinary differential equations can be solved numerically with `scipy.integrate.odeint`. This function is based on the well-tested Fortran LSODA routine, which can automatically switch between stiff and nonstiff algorithms.[8] `odeint` solves first-order differential equations – *to solve a higher-order equation, it must be decomposed into a system of first-order equations first*, as explained later.

A single first-order ordinary differential equation

In its simplest use for the solution of a single first-order ordinary differential equation,

$$\frac{\mathrm{d}y}{\mathrm{d}t} = f(y, t),$$

`odeint` takes three arguments: a function object returning $\mathrm{d}y/\mathrm{d}t$, an initial condition, y_0, and a sequence of t values at which to calculate the solution, $y(t)$.

For example, consider the first-order differential equation describing the rate of the reaction A → P in terms of the concentration of the reactant, A:

$$\frac{\mathrm{d}[\mathrm{A}]}{\mathrm{d}t} = -k[\mathrm{A}].$$

This example has an easily obtainable analytical solution:

$$[\mathrm{A}] = [\mathrm{A}]_0 e^{-kt},$$

where $[\mathrm{A}]_0$ is the initial concentration of [A].

[8] A differential equation is said to be *stiff* if a numerical method is required to take excessively small steps in its intervals of integration in relation to the smoothness of the exact underlying solution.

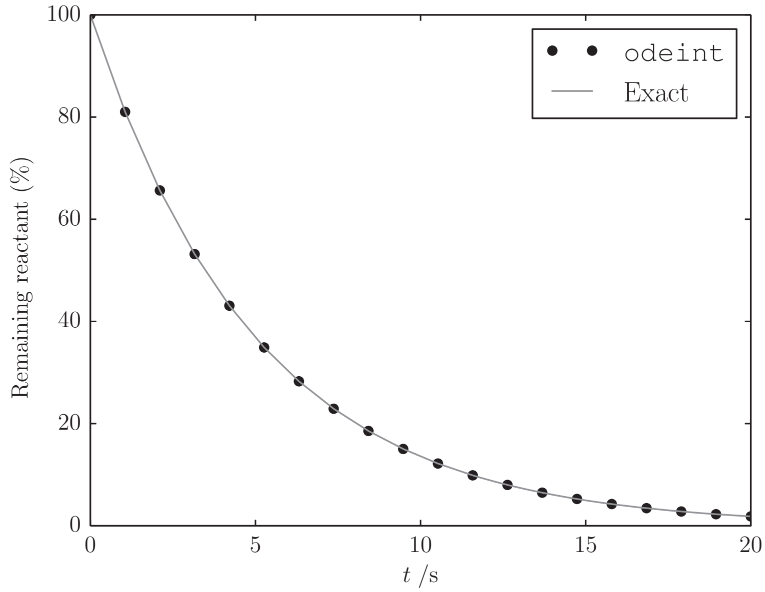


Figure 8.9 Exponential decay of a reactant in a first-order reaction: numerical and exact solutions.

To solve the equation numerically with `odeint`, write it in the form as shown above, with a single dependent variable, $y(t) \equiv [A]$, which is a function of the independent variable, t (time). We have:

$$\frac{dy}{dt} = -ky$$

We need to provide a function returning dy/dt as $f(y, t)$ (in general a function of both y and t), an initial condition, $y(0)$ and a sequence of time points upon which to calculate the solution. The derivative function is simply:

```
def dydt(y, t):
    return -k * y
```

(the order of the arguments is important). A program comparing the numerical and analytical results for a reaction with $k = 0.2\ s^{-1}$ and $y(0) \equiv [A]_0 = 100$ is given later; the resulting plot is Figure 8.9.

Listing 8.9 First-order reaction kinetics

```
import numpy as np
from scipy.integrate import odeint
import pylab

# First-order reaction rate constant, s-1
k = 0.2
# Initial condition on y: 100% of reactant is present at t=0
y0 = 100

# A suitable grid of time points for the reaction
t = np.linspace(0, 20, 20)
```

```
def dydt(y, t):
    """ Return dy/dt = f(y,t) at time t. """
    return -k * y

# Integrate the differential equation
y = odeint(dydt, y0, t)

# Plot and compare the numerical and exact solutions
pylab.plot(t, y, 'o', color='k', label=r'\texttt{odeint}')
pylab.plot(t, y0 * np.exp(-k*t), color='gray', label='Exact')
pylab.xlabel(r'$t\;/\mathrm{s}$')
pylab.ylabel('Remaining reactant (\%)')
pylab.legend()
pylab.show()
```

As with the `quad` family of routines, if the function returning the derivative requires further arguments, they can be passed to `odeint` in the `args` parameter. In the earlier mentioned example, `k` is resolved in global scope, but we could pass it with:

```
def dydt(y, t, k):
    return -k * y
```

(note that additional parameters must appear after the dependent and independent variables). The call to `odeint` would then be:

```
y = odeint(dydt, y0, t, args=(k,))
```

Coupled first-order ordinary differential equations

`odeint` can also solve a set of coupled first-order differential equations in more than one dependent variable: $y_1(t), y_2(t), \cdots, y_n(t)$:

$$
\begin{aligned}
\frac{\mathrm{d}y_1}{\mathrm{d}t} &= f_1(y_1, y_2, \cdots, y_n; t)\\
\frac{\mathrm{d}y_2}{\mathrm{d}t} &= f_2(y_1, y_2, \cdots, y_n; t)\\
&\cdots\\
\frac{\mathrm{d}y_n}{\mathrm{d}t} &= f_n(y_1, y_2, \cdots, y_n; t)
\end{aligned}
$$

In this case, the function passed to `odeint()` must return a sequence of derivatives, $\mathrm{d}y_1/\mathrm{d}t, \mathrm{d}y_2/\mathrm{d}t, \cdots, \mathrm{d}y_n/\mathrm{d}t$ for each of the dependent variables; that is, it evaluates the earlier mentioned functions $f_i(y_1, y_2, \cdots, y_n; t)$ for each of the y_i passed to it in a sequence, `y`. The form of this function is:

```
def deriv(y, t):
    # y = [y1, y2, y3, ...] is a sequence of dependent variables
    dy1dt = f1(y, t)     # calculate dy1/dt as f1(y1,y2,...,yn;t)
    dy2dt = f2(y, t)     # calculate dy2/dt as f2(y1,y2,...,yn;t)
    # ... etc
    # Return the derivatives in a sequence such as a tuple:
    return dy1dt, dy2dt, ..., dyndt
```

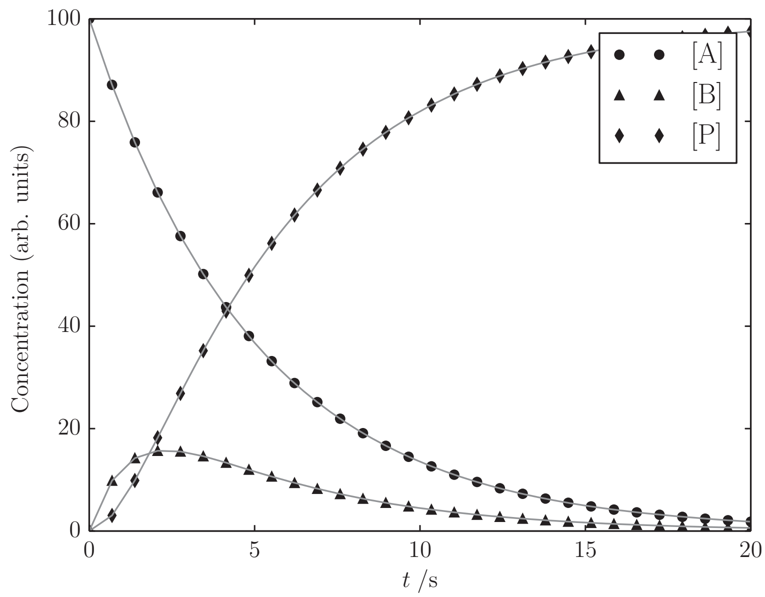


Figure 8.10 Two coupled first-order reactions: numerical and exact solutions.

For a concrete example, suppose a reaction proceeds via two first-order reaction steps: A → B → P with rate constants k_1 and k_2. The equations governing the rate of change of A and B are

$$\frac{\mathrm{d[A]}}{\mathrm{d}t} = -k_1[\mathrm{A}]$$
$$\frac{\mathrm{d[B]}}{\mathrm{d}t} = k_1[\mathrm{A}] - k_2[\mathrm{B}]$$

Again, we can solve this pair of coupled equations analytically, but in our numerical solution, let $y_1 \equiv [\mathrm{A}]$ and $y_2 \equiv [\mathrm{B}]$:

$$\frac{\mathrm{d}y_1}{\mathrm{d}t} = -k_1 y_1$$
$$\frac{\mathrm{d}y_2}{\mathrm{d}t} = k_1 y_1 - k_2 y_2$$

The code mentioned here integrates these equations for $k_1 = 0.2\ \mathrm{s}^{-1}, k_2 = 0.8\ \mathrm{s}^{-1}$ and initial conditions $y_1(0) = 100$, $y_2(0) = 0$, and compares with the analytical result (Figure 8.10).

Listing 8.10 Two coupled first-order reactions

```
import numpy as np
from scipy.integrate import odeint
import pylab

# First-order reaction rate constants, s-1
k1, k2 = 0.2, 0.8
# Initial condition on y1, y2: [A](t=0) = 100, [B](t=0) = 0
A0, B0 = 100, 0
```

```
# A suitable grid of time points for the reaction
t = np.linspace(0, 20, 100)

def dydt(y, t, k1, k2):
    """ Return dy_i/dt = f(y_i,t) at time t. """
    y1, y2 = y
    dy1dt = -k1 * y1
    dy2dt = k1 * y1 - k2 * y2
    return dy1dt, dy2dt

# Integrate the differential equation
y0 = A0, B0
y1, y2 = odeint(dydt, y0, t, args=(k1, k2)).T   ❶

A, B = y1, y2
# [P] is determined by conservation
P = A0 - A - B

# Analytical result
Aexact = A0 * np.exp(-k1*t)
Bexact = A0 * k1/(k2-k1) * (np.exp(-k1*t) - np.exp(-k2*t))
Pexact = A0 - Aexact - Bexact

pylab.plot(t, A, 'o', label='[A]')
pylab.plot(t, B, '^', label='[B]')
pylab.plot(t, P, 'd', label='[P]')
pylab.plot(t, Aexact)
pylab.plot(t, Bexact)
pylab.plot(t, Pexact)
pylab.xlabel(r'$t\;/\mathrm{s}$')
pylab.ylabel('Concentration (arb. units)')
pylab.legend()
pylab.show()
```

❶ Note that `odeint` returns a two-dimensional array with the values of each dependent variable in the *rows*: if we want to unpack this array to separate one-dimensional arrays, `y1`, `y2`, and so on, we need the transpose of this returned array.

A single second-order ordinary differential equation

To solve an ordinary differential equation of higher than first order, it must first be reduced into a system of first-order differential equations. In general, any differential equation with a single dependent variable of order n can be written as a system of n first-order differential equations in n dependent variables.

For example, the equation of motion for a harmonic oscillator is a second-order differential equation:

$$\frac{\mathrm{d}^2x}{\mathrm{d}t^2} = -\omega^2 x,$$

where x is the displacement from equilibrium and ω is the angular frequency. This equation may be decomposed into two first-order equations as follows:

$$\frac{dx_1}{dt} = x_2,$$
$$\frac{dx_2}{dt_2} = -\omega^2 x_1,$$

where x_1 is identified with x and x_2 with dx/dt.

This pair of coupled first-order equations may be solved as before:

Listing 8.11 Solution of the harmonic oscillator equation of motion

```
import numpy as np
from scipy.integrate import odeint
import pylab

# Harmonic oscillator frequency (s-1)
omega = 0.9
# initial conditions on x1=x and x2=dx/dt at t=0
A, v0 = 3, 0        # cm, cm.s-1
x0 = A, v0

# A suitable grid of time points
t = np.linspace(0, 20, 100)

def dxdt(x, t, omega):
    """ Return dx/dt = f(x,t) at time t. """
    x1, x2 = x
    dx1dt = x2
    dx2dt = -omega**2 * x1
    return dx1dt, dx2dt

# Integrate the differential equation
x1, x2 = odeint(dxdt, x0, t, args=(omega,)).T

# Plot and compare the numerical and exact solutions
pylab.plot(t, x1, 'o', color='k', label=r'\texttt{odeint()}')
pylab.plot(t, A * np.cos(omega * t), color='gray', label='Exact')
pylab.xlabel(r'$t\;/\mathrm{s}$')
pylab.ylabel(r'$x\;/\mathrm{cm}$')
pylab.legend()
pylab.show()
```

The plot produced by this code is given in Figure 8.10.

The `odeint` function is a simplified interface to the more advanced `scipy.integrate.ode` method which provides a range of different numerical integrators, including Runge-Kutta algorithms and support for complex-valued variables.

Example E8.16 An object falling slowly in a viscous fluid under the influence of gravity is subject to a drag force (*Stokes drag*), which varies linearly with its velocity. Its equation of motion may be written as the second-order differential equation:

$$m\frac{d^2z}{dt^2} = -c\frac{dz}{dt} + mg',$$

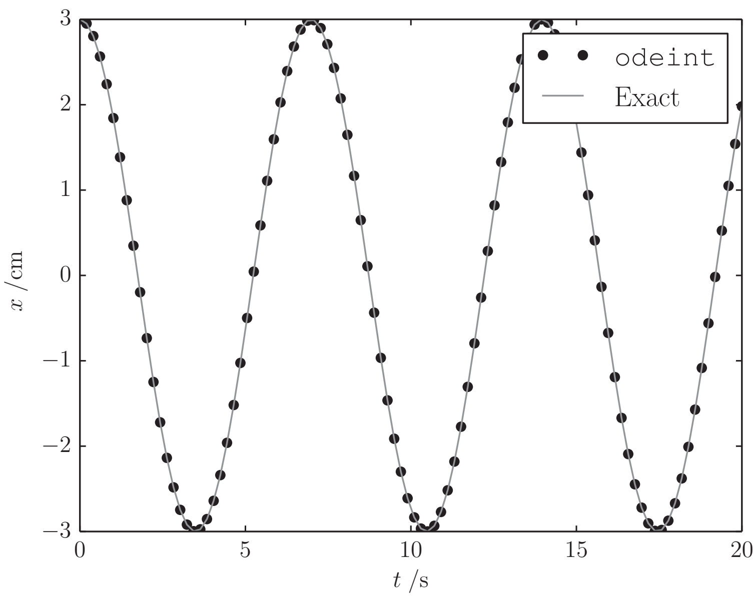


Figure 8.11 The harmonic oscillator: numerical and exact solutions.

where z is the object's position as a function of time, t, c is a drag constant which depends on the shape of the object and the fluid viscosity and

$$g' = g\left(1 - \frac{\rho_{\text{fluid}}}{\rho_{\text{obj}}}\right)$$

is the effective gravitational acceleration, which accounts for the buoyant force due to the fluid (density ρ_{fluid}) displaced by the object (density ρ_{obj}). For a small sphere of radius r in a fluid of viscosity η, Stokes' law predicts $c = 6\pi\eta r$.

Consider a sphere of platinum ($\rho = 21.45\ \text{g cm}^{-3}$) with radius 1 mm, initially at rest, falling in mercury ($\rho = 13.53\ \text{g cm}^{-3}$, $\eta = 1.53 \times 10^{-3}$ Pa s). The earlier mentioned second-order differential equation can be solved analytically, but to integrate it numerically using `odeint`, it must be treated as two first-order ordinary differential equations:

$$\frac{\mathrm{d}z}{\mathrm{d}t} = \dot{z}$$
$$\frac{\mathrm{d}^2 z}{\mathrm{d}t^2} = \frac{\mathrm{d}\dot{z}}{\mathrm{d}t} = g' - \frac{c}{m}\dot{z}$$

In the code mentioned here, the function `deriv` calculates these derivatives and is passed to `odeint` with the intial conditions ($z = 0$, $\dot{z} = 0$) and a grid of time points.

Listing 8.12 Calculating the motion of a sphere falling under the influence of gravity and Stokes drag

```
# eg8-stokes-drag.py
import numpy as np
from scipy.integrate import odeint
import pylab

# Pt sphere falling from rest in mercury
```

```
# Acceleration due to gravity (m.s-2)
g = 9.81
# Densities (kg.m-3)
rho_Pt, rho_Hg = 21450, 13530
# Viscosity  of Hg (Pa.s)
eta = 1.53e-3

# Radius and mass of the sphere
r = 1.e-3   # radius (m)
m = 4*np.pi/3 * r**3 * rho_Pt
# Drag consant from Stokes' Law:
c = 6 * np.pi * eta * r
# Effective gravitational acceleration
gp = g * (1 - rho_Hg/rho_Pt)

def deriv(z, t, m, c, gp):
    """ Return the dz/dt and d2z/dt2. """
    dz0 = z[1]
    dz1 = gp - c/m * z[1]
    return dz0, dz1

t = np.linspace(0, 20, 50)
# Initial conditions: z = 0, dz/dt = 0 at t=0
z0 = (0, 0)

# Integrate the pair of differential equations
z, zdot = odeint(deriv, z0, t, args=(m, c, gp)).T
pylab.plot(t, zdot)

print('Estimate of terminal velocity = {:.3f} m.s-1'.format(zdot[-1]))

# Exact solution: terminal velocity vt (m.s-1) and characteristic time tau (s)
v0, vt, tau = 0, m*gp/c, m/c
print('Exact terminal velocity = {:.3f} m.s-1'.format(vt))
z = vt*t + v0*tau*(1-np.exp(-t/tau)) + vt*tau*(np.exp(-t/tau)-1)
zdot_exact = vt + (v0-vt)*np.exp(-t/tau)
pylab.plot(t, zdot_exact)
pylab.xlabel('$t$ /s')
pylab.ylabel('$\dot{z}\;/\mathrm{m\, s^{-1}}$')

pylab.show()
```

The plot produced by this program is shown in Figure 8.12: the numerical and analytical results are indistinguishable at this scale but are reported to three decimal places in the output:

```
Estimate of terminal velocity = 11.266 m.s-1
Exact terminal velocity = 11.285 m.s-1
```

8.2.4 Exercises

Questions

Q8.2.1 Use `scipy.integrate.quad` to evaluate the following integral:

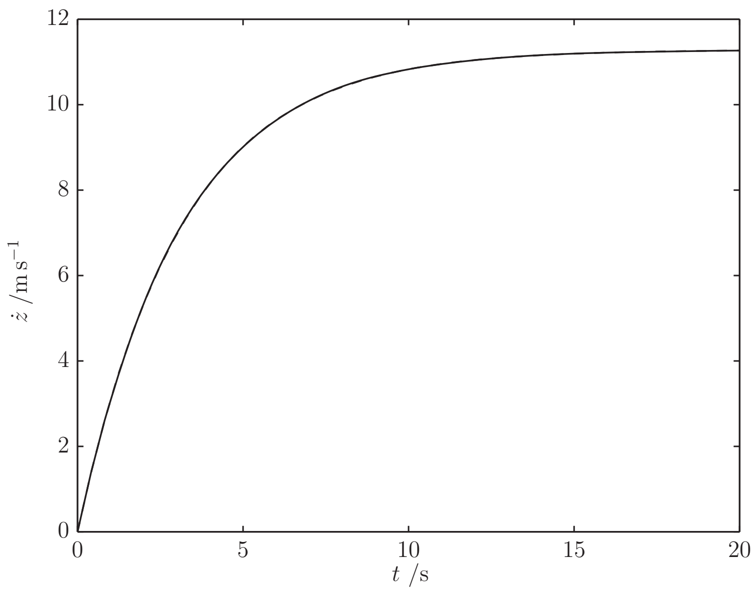


Figure 8.12 The velocity of a platinum sphere falling in mercury as a function of time, modeled with Stokes' law.

$$\int_0^6 \lfloor x \rfloor - 2\left\lfloor \frac{x}{2} \right\rfloor \,\mathrm{d}x.$$

Q8.2.2 Use `scipy.integrate.quad` to evaluate the following definite integrals (most of which can also be expressed in closed form over the range given but are awkward).

a.

$$\int_0^1 \frac{x^4(1-x)^4}{1+x^2}\,\mathrm{d}x.$$

(Compare with $22/7 - \pi$.)

b. The following integral appears in the Debye theory of the heat capacity of crystals at low temperature

$$\int_0^\infty \frac{x^3}{e^x - 1}\,\mathrm{d}x.$$

(Compare with $\pi^4/15$.)

c. The integral sometimes known as the *Sophomore's dream*:

$$\int_0^1 x^{-x}\,\mathrm{d}x$$

(Compare the value you obtain from the summation $\sum_{n=1}^{\infty} n^{-n}$.)

d.

$$\int_0^1 [\ln(1/x)]^p\,\mathrm{d}x$$

(Compare with $p!$ for integer $0 \le p \le 10$.)

e.

$$\int_0^{2\pi} e^{z\cos\theta}\,\mathrm{d}\theta$$

(Compare with $I_0(z)/2\pi$, where $I_0(z)$ is a modified Bessel function of the first kind, for $0 \le z \le 2$.)

Q8.2.3 Use `scipy.integrate.dblquad` to evaluate π by integration of the constant function $f(x, y) = 4$ over the quarter circle with unit radius in the quadrant $x > 0, y > 0$.

Q8.2.4 What is wrong with the following attempt to calculate the area of the unit circle (π) as a double integral in polar coordinates?

```
In [x]: dblquad(lambda r, theta: r, 0, 1, lambda r: 0, lambda r: 2*np.pi)
Out[x]: (19.739208802178712, 2.1914924100062363e-13)
```

Problems

P8.2.1 The area of the surface of revolution about the x-axis between a and b of the function $y = f(x)$ is given by the integral

$$S = 2\pi\int_a^b y\,\mathrm{d}s, \quad \text{where } \mathrm{d}s = \sqrt{1+\left(\frac{\mathrm{d}y}{\mathrm{d}x}\right)^2}\,\mathrm{d}x.$$

Use this equation to write a function to determine the surface area of revolution of a function $y = f(x)$ about the x-axis, given Python function objects that return y and $\mathrm{d}y/\mathrm{d}x$, and test it for the paraboloid obtained by rotation of the function $f(x) = \sqrt{x}$ about the x-axis between $a = 0$ and $b = 1$. Compare with the exact result, $\pi(5^{3/2} - 1)/6$.

P8.2.2 The integral of the secant function,

$$\int_0^{\theta} \sec\phi\,\mathrm{d}\phi$$

for $-\pi/2 < \theta < \pi/2$ is important in navigation and the theory of map projections. It can be expressed in closed form as the inverse Gudermannian function,

$$\mathrm{gd}^{-1}(\theta) = \ln|\sec\theta + \tan\theta|.$$

Use `scipy.integrate.quad` to calculate values for the integral across the relevant range for θ given earlier and compare graphically with the exact answer.

P8.2.3 Consider a torus of uniform density, unit mass, average radius R and cross-sectional radius r. The volume and moments of inertia of such a torus may be evaluated analytically and give the results:

$$V = 2\pi^2 R r^2,$$
$$I_z = R^2 + \tfrac{3}{4}r^2,$$
$$I_x = I_y = \tfrac{1}{2}R^2 + \tfrac{5}{8}r^2,$$

where the center of mass of the torus is at the origin and the z axis is taken to be its symmetry axis.

Here we take a numerical approach. In cylindrical coordinates (ρ, θ, z), it may be shown that:

$$V = 2\int_0^{2\pi}\int_{R-r}^{R+r}\int_0^{\sqrt{r^2-(\rho-R)^2}} \rho \, \mathrm{d}z\,\mathrm{d}\rho\,\mathrm{d}\theta,$$

$$I_z = \frac{2}{V}\int_0^{2\pi}\int_{R-r}^{R+r}\int_0^{\sqrt{r^2-(\rho-R)^2}} \rho^3 \, \mathrm{d}z\,\mathrm{d}\rho\,\mathrm{d}\theta,$$

$$I_x = I_y = \frac{2}{V}\int_0^{2\pi}\int_{R-r}^{R+r}\int_0^{\sqrt{r^2-(\rho-R)^2}} (\rho^2\sin^2\theta + z^2)\rho \, \mathrm{d}z\,\mathrm{d}\rho\,\mathrm{d}\theta.$$

Evaluate these integrals for the torus with dimensions $R = 4$, $r = 1$ and compare with the exact values.

P8.2.4 The *Brusselator* is a theoretical model for an autocatalytic reaction. It assumes the following reaction sequence, in which species A and B are taken to be in excess with constant concentration and species D and E are removed as they are produced. The concentrations of species X and Y can show oscillatory behavior under certain conditions.

$$\begin{aligned} \mathrm{A} &\rightarrow \mathrm{X} & k_1 \\ 2\mathrm{X} + \mathrm{Y} &\rightarrow 3\mathrm{X} & k_2 \\ \mathrm{B} + \mathrm{X} &\rightarrow \mathrm{Y} + \mathrm{D} & k_3 \\ \mathrm{X} &\rightarrow \mathrm{E} & k_4 \end{aligned}$$

It is convenient to introduce the scaled quantities

$$x = [\mathrm{X}]\sqrt{\frac{k_2}{k_4}}, \quad y = [\mathrm{Y}]\sqrt{\frac{k_2}{k_4}},$$

$$a = [\mathrm{A}]\frac{k_1}{k_4}\sqrt{\frac{k_2}{k_4}}, \quad b = [\mathrm{B}]\frac{k_3}{k_4},$$

and to scale the time by the factor k_4, which gives rise to the dimensionless equations

$$\frac{\mathrm{d}x}{\mathrm{d}t} = a - (1+b)x + x^2y,$$

$$\frac{\mathrm{d}y}{\mathrm{d}t} = bx + x^2y.$$

Show how these equations predict x and y to vary for (a) $a = 1, b = 1.8$ and (b) $a = 1, b = 2.02$ by plotting in each case (i) x, y as functions of (dimensionless) time and (ii) y as a function of x.

P8.2.5 The equation governing the motion of a pendulum consisting of a mass at the end of a light, rigid rod of length l may be written

$$\frac{\mathrm{d}^2\theta}{\mathrm{d}t^2} = -\frac{g}{l}\sin\theta,$$

where θ is the angle the pendulum makes with the vertical.

Taking $l = 1$ m and $g = 9.81$ m s^{-2}, determine the subsequent motion of the pendulum if it is started at rest with an initial angle $\theta_0 = 30°$. Compare the motion with the harmonic approximation reached by assuming θ is small, which has the analytical solution $\theta = \theta_0\cos(\omega t)$ with $\omega = \sqrt{g/l}$.

P8.2.6 A simple mechanism for the formation of ozone in the stratosphere consists of the following four reactions (known as the *Chapman cycle*):

$$\begin{aligned}
&\mathrm{O_2} + h\nu \rightarrow 2\mathrm{O} && k_1 = 3\times10^{-12}\ \mathrm{s^{-1}}\\
&\mathrm{O_2} + \mathrm{O} + \mathrm{M} \rightarrow \mathrm{O_3} + \mathrm{M} && k_2 = 1.2\times10^{-33}\ \mathrm{cm^6\,molec^{-2}\,s^{-1}}\\
&\mathrm{O_3} + h\nu' \rightarrow \mathrm{O} + \mathrm{O_2} && k_3 = 5.5\times10^{-4}\ \mathrm{s^{-1}}\\
&\mathrm{O} + \mathrm{O_3} \rightarrow 2\mathrm{O_2} && k_4 = 6.9\times10^{-16}\ \mathrm{cm^3\,molec^{-1}\,s^{-1}}
\end{aligned}$$

where M is a nonreacting third body taken to be at the total air molecule concentration for the altitude being considered. The earlier mentioned reactions lead to the following rate equations for [O], $[O_3]$ and $[O_2]$:

$$\begin{aligned}
\frac{\mathrm{d}[\mathrm{O_2}]}{\mathrm{d}t} &= -k_1[\mathrm{O_2}] - k_2[\mathrm{O_2}][\mathrm{O}][\mathrm{M}] + k_3[\mathrm{O_3}] + 2k_4[\mathrm{O}][\mathrm{O_3}]\\
\frac{\mathrm{d}[\mathrm{O}]}{\mathrm{d}t} &= 2k_1[\mathrm{O_2}] - k_2[\mathrm{O_2}][\mathrm{O}][\mathrm{M}] + k_3[\mathrm{O_3}] - k_4[\mathrm{O}][\mathrm{O_3}]\\
\frac{\mathrm{d}[\mathrm{O_3}]}{\mathrm{d}t} &= k_2[\mathrm{O_2}][\mathrm{O}][\mathrm{M}] - k_3[\mathrm{O_3}] - k_4[\mathrm{O}][\mathrm{O_3}]
\end{aligned}$$

The rate constants apply at an altitude of 25 km, where $[\mathrm{M}] = 9\times10^{17}$ molec cm^{-3}. Write a program to determine the concentrations of O_3 and O as a function of time at this altitude (you should find the $[O_2]$ remains pretty much constant). Start with initial conditions $[\mathrm{O_2}]_0 = 0.21[\mathrm{M}]$, $[\mathrm{O}]_0 = [\mathrm{O_3}]_0 = 0$ and integrate for 10^8 s (starting from scratch it takes about three years to build an ozone layer with this mechanism). Compare the equilibrium concentrations with the approximate analytical result obtained using the *steady-state approximation*:

$$[\mathrm{O_3}] = \sqrt{\frac{k_1k_2}{k_3k_4}}[\mathrm{O_2}][\mathrm{M}]^{\frac{1}{2}}, \quad \frac{[\mathrm{O}]}{[\mathrm{O_3}]} = \frac{k_3}{k_2[\mathrm{O_2}][\mathrm{M}]}.$$

P8.2.7 Hyperion is an irregularly shaped moon of Saturn notable for its chaotic rotation. Its motion may be modeled as follows.

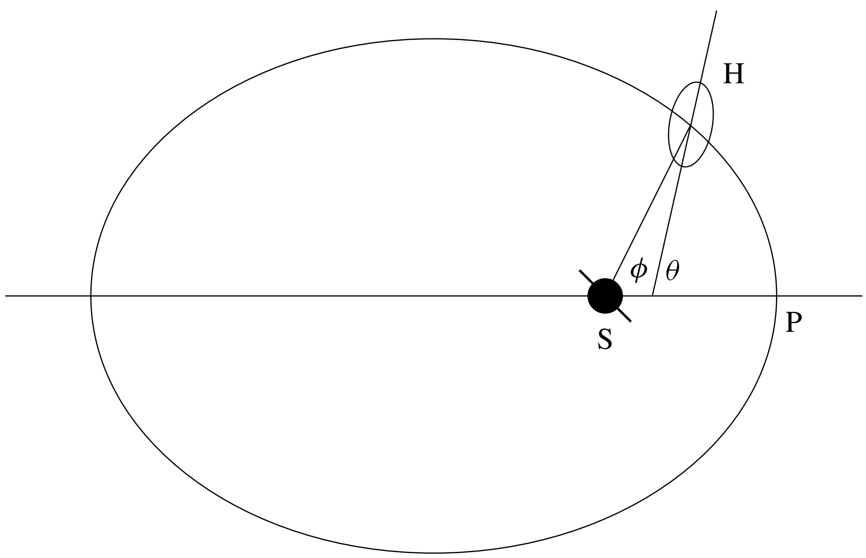
The orbit of Hyperion (H) about Saturn (S) is an ellipse with semi-major axis, a, and eccentricity, e. Let its point of closest approach (*periapsis*) be P. Its distance from the

planet, SH, as a function of its *true anomaly* (orbital angle, ϕ, measured from the line SP) is therefore

$$r = \frac{a(1-e^2)}{1+e\cos\phi}.$$

Define the angle θ to be that between the axis of the smallest principal moment of inertia (loosely, the longest axis of the moon) and SP, and the quantity Ω to be a scaled rate of change of θ with ϕ (i.e., the rate at which Hyperion spins as it orbits Saturn) as follows:

$$\Omega = \frac{a^2}{r^2}\frac{d\theta}{d\phi}.$$



Now, it can be shown that

$$\frac{d\Omega}{d\phi} = -\frac{B-A}{C}\frac{3}{2(1-e^2)}\frac{a}{r}\sin[2(\theta-\phi)],$$

where A, B and C are the principal moments of inertia.

Use `scipy.integrate.odeint` to find and plot the spin rate, Ω, as a function of ϕ for the initial conditions (a) $\theta = \Omega = 0$ at $\phi = 0$, and (b) $\theta = 0$, $\Omega = 2$ at $\phi = 0$. Take $e = 0.1$ and $(B-A)/C = 0.265$.

P8.2.8 The radioactive decay chain of ^{212}Pb to the stable isotope ^{208}Pb may be considered as the following sequence of steps with the given rate constants, k_i:

$$\begin{aligned}
^{212}\text{Pb} &\rightarrow {}^{212}\text{Bi} + \beta^- & k_1 &= 1.816\times10^{-5}\ \text{s}^{-1},\\
^{212}\text{Bi} &\rightarrow {}^{208}\text{Tl} + \alpha & k_2 &= 6.931\times10^{-5}\ \text{s}^{-1},\\
^{212}\text{Bi} &\rightarrow {}^{212}\text{Po} + \beta^- & k_3 &= 1.232\times10^{-4}\ \text{s}^{-1},\\
^{208}\text{Tl} &\rightarrow {}^{208}\text{Pb} + \beta^- & k_4 &= 3.851\times10^{-3}\ \text{s}^{-1},\\
^{212}\text{Po} &\rightarrow {}^{208}\text{Pb} + \alpha & k_5 &= 2.310\ \text{s}^{-1}.
\end{aligned}$$

By considering the following first-order differential equations giving the rates of change for each species, plot their concentrations as a function of time.

$$\frac{\mathrm{d}[^{212}\mathrm{Pb}]}{\mathrm{d}t} = -k_1[^{212}\mathrm{Pb}]$$
$$\frac{\mathrm{d}[^{212}\mathrm{Bi}]}{\mathrm{d}t} = k_1[^{212}\mathrm{Pb}] - k_2[^{212}\mathrm{Bi}] - k_3[^{212}\mathrm{Bi}]$$
$$\frac{\mathrm{d}[^{208}\mathrm{Tl}]}{\mathrm{d}t} = k_2[^{212}\mathrm{Bi}] - k_4[^{208}\mathrm{Tl}]$$
$$\frac{\mathrm{d}[^{212}\mathrm{Po}]}{\mathrm{d}t} = k_3[^{212}\mathrm{Bi}] - k_5[^{212}\mathrm{Po}]$$
$$\frac{\mathrm{d}[^{208}\mathrm{Pb}]}{\mathrm{d}t} = k_4[^{208}\mathrm{Tl}] + k_5[^{212}\mathrm{Po}]$$

If all the intermediate species, J, are treated in "steady state" (i.e., $\mathrm{d}[\mathrm{J}]/\mathrm{d}t = 0$, the approximate expression for the ^{208}Pb concentration as a function of time is

$$[^{208}\mathrm{Pb}] = [^{212}\mathrm{Pb}]_0\left(1 - e^{-k_1 t}\right).$$

Compare the "exact" result obtained by numerical integration of the differential equations with this approximate answer.

8.3 Interpolation

The package `scipy.interpolate` contains a large variety of functions and classes for interpolation and splines in one and more dimensions. Some of the more important are described in this section.

8.3.1 Univariate interpolation

The most straightforward one-dimensional interpolation functionality is provided by `scipy.interpolate.interp1d`. Given arrays of points `x` and `y`, a function is returned, which can be called to generate interpolated values at intermediate values of `x`. The default interpolation scheme is linear, but other options (see Table 8.3) allow for different schemes, as shown in the following example.

Example E8.17 This example demonstrates some of the different interpolation methods available in `scipy.interpolation.interp1d` (see Figure 8.13).

Listing 8.13 A comparison of one-dimensional interpolation types using `scipy.interpolate.interp1d`

```
# eg8-interp1d.py
import numpy as np
from scipy.interpolate import interp1d
import pylab
```

Table 8.3 Interpolation methods specified by the `kind` argument to `scipy.interpolate.interp1d`

kind	Description
'linear'	The default, linear interpolation using only the values from the original data arrays bracketing the desired point
'nearest'	"Snap" to the nearest data point
'zero'	A zeroth-order spline: interpolates to the last value seen in its traversal of the data arrays
'slinear'	First-order spline interpolation (in practice, the same as 'linear')
'quadratic'	Second-order spline interpolation
'cubic'	Cubic spline interpolation

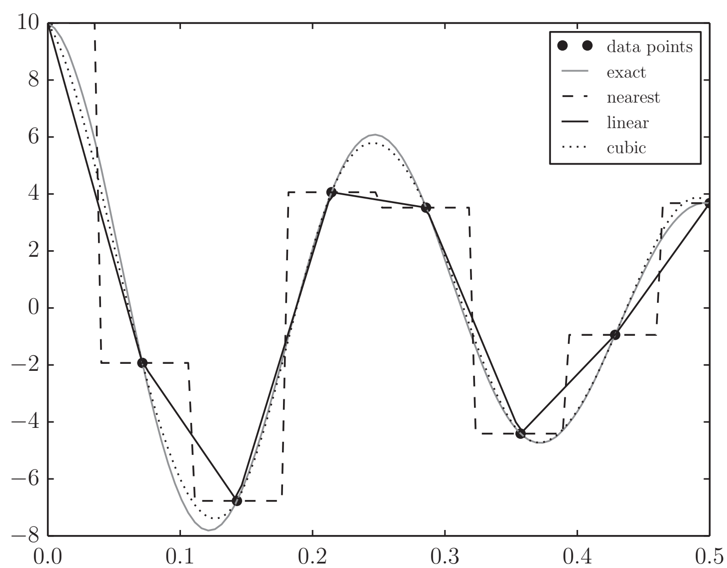


Figure 8.13 An illustration of different one-dimensional interpolation methods with `scipy.interpolation.interp1d`.

```
A, nu, k = 10, 4, 2

def f(x, A, nu, k):
    return A * np.exp(-k*x) * np.cos(2*np.pi * nu * x)

xmax, nx = 0.5, 8
x = np.linspace(0, xmax, nx)
y = f(x, A, nu, k)

f_nearest = interp1d(x, y, kind='nearest')
f_linear  = interp1d(x, y)
f_cubic   = interp1d(x, y, kind='cubic')

x2 = np.linspace(0, xmax, 100)
pylab.plot(x, y, 'o', label='data points')
pylab.plot(x2, f(x2, A, nu, k), label='exact')
pylab.plot(x2, f_nearest(x2), label='nearest')
```

```
pylab.plot(x2, f_linear(x2), label='linear')
pylab.plot(x2, f_cubic(x2), label='cubic')
pylab.legend()
pylab.show()
```

8.3.2 Multivariate interpolation

We shall consider two kinds of multivariate interpolation corresponding to whether or not the source data are structured (arranged on some kind of grid) or not.

Interpolation from a rectangular grid

The simplest two-dimensional interpolation routine is `scipy.interpolate.interp2d`. It requires a two-dimensional array of values, `z`, and the two (one-dimensional) coordinate arrays `x` and `y` to which they correspond. These arrays need not have constant spacing. Three kinds of interpolation spline are supported through the `kind` argument: `'linear'` (the default), `'cubic'` and `'quintic'`.

Example E8.18 In the following example, we calculate the function

$$z(x, y) = \sin\left(\frac{\pi x}{2}\right) e^{y/2}$$

on a grid of points (x, y) which is not evenly spaced in the y-direction. We then use `scipy.interpolate.interp2d` to interpolate these values onto a finer, evenly spaced (x, y) grid: see Figure 8.14.

Listing 8.14 Two-dimensional interpolation with `scipy.interpolate.interp2d`

```
# eg8-interp2d.py
import numpy as np
from scipy.interpolate import interp2d
import matplotlib.pyplot as plt

x = np.linspace(0, 4, 13)
y = np.array([0, 2, 3, 3.5, 3.75, 3.875, 3.9375, 4])
X, Y = np.meshgrid(x, y)
Z = np.sin(np.pi*X/2) * np.exp(Y/2)

x2 = np.linspace(0, 4, 65)
y2 = np.linspace(0, 4, 65)
f = interp2d(x, y, Z, kind='cubic')    ❶
Z2 = f(x2, y2)

fig, ax = plt.subplots(nrows=1, ncols=2)
ax[0].pcolormesh(X, Y, Z)

X2, Y2 = np.meshgrid(x2, y2)
ax[1].pcolormesh(X2, Y2, Z2)

plt.show()
```

❶ Note that `interp2d` requires the *one-dimensional* arrays, `x` and `y`.

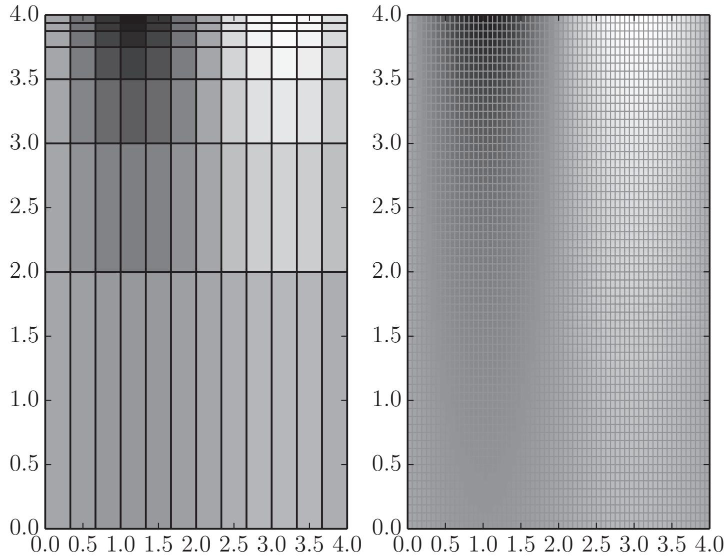


Figure 8.14 Two-dimensional interpolation with `scipy.interpolate.interp2d`.

If the mesh of (x, y) coordinates form a *regularly spaced* grid, the fastest way to interpolate values from values of `z` is to use a `scipy.interpolate.RectBivariateSpline` object as in the following example.

Example E8.19 In the following code, the function

$$z(x, y) = e^{-4x^2} e^{-y^2/4}$$

is calculated on a regular, coarse grid and then interpolated onto a finer one (Figure 8.15).

Listing 8.15 Interpolation onto a regular two-dimensional grid with `scipy.interpolate.RectBivariateSpline`

```
# eg8-RectBivariateSpline.py
import numpy as np
from scipy.interpolate import RectBivariateSpline
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

# Regularly spaced, coarse grid
dx, dy = 0.4, 0.4
xmax, ymax = 2, 4
x = np.arange(-xmax, xmax, dx)
y = np.arange(-ymax, ymax, dy)
X, Y = np.meshgrid(x, y)
Z = np.exp(-(2*X)**2 - (Y/2)**2)

❶ interp_spline = RectBivariateSpline(y, x, Z)

# Regularly spaced, fine grid
dx2, dy2 = 0.16, 0.16
x2 = np.arange(-xmax, xmax, dx2)
```

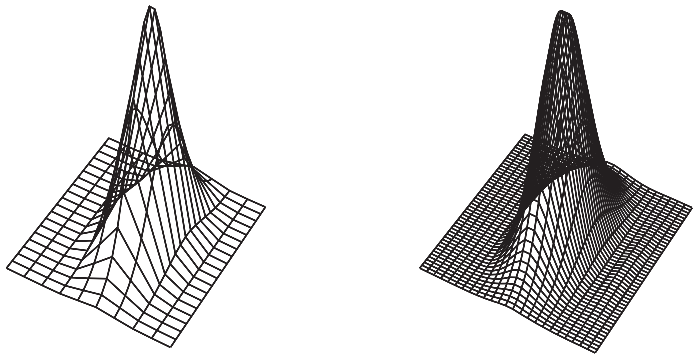


Figure 8.15 Two-dimensional interpolation from a coarse rectangular grid (left-hand plot) to a finer one (right-hand plot) with `scipy.interpolate.RectBivariateSpline`.

```
y2 = np.arange(-ymax, ymax, dy2)
X2, Y2 = np.meshgrid(x2,y2)
Z2 = interp_spline(y2, x2)

fig, ax = plt.subplots(nrows=1, ncols=2, subplot_kw={'projection': '3d'})
ax[0].plot_wireframe(X, Y, Z, color='k')

ax[1].plot_wireframe(X2, Y2, Z2, color='k')
for axes in ax:
    axes.set_zlim(-0.2,1)
    axes.set_axis_off()

fig.tight_layout()
plt.show()
```

❶ Note that for our function, `Z`, defined using the `meshgrid` set up here, the `RectBivariateSpline` method expects the corresponding one-dimensional arrays `y` and `x` to be passed in this order (opposite to that of `interp2d`).[9]

Interpolation of unstructured data

To interpolate unstructured data (that is, data points provided at *arbitrary* coordinates (x, y)) onto a grid, the method `scipy.interpolate.griddata` can be used. Its basic usage for two dimensions is:

```
scipy.interpolate.griddata(points, values, xi, method='linear')
```

where the provided data are given as the one-dimensional array, `values`, at the coordinates `points`, which is provided as a tuple of arrays `x` and `y` or as a single array of shape `(n,2)` where `n` is the length of the `values` array. `xi` is an array of the coordinate grid to by interpolated onto (of shape `(m,2)`.) The `methods` available are `'linear'` (the default), `'nearest'` and `'cubic'`.

[9] This issue is related to the way that `meshgrid` is indexed, which is based on the conventions of MATLAB.

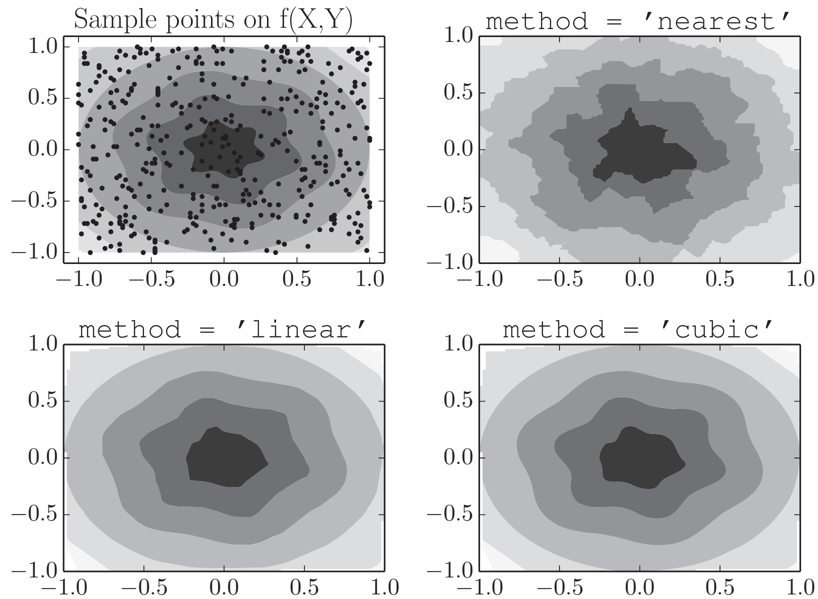


Figure 8.16 Some different interpolation schemes for `scipy.interpolate.griddata`.

Example E8.20 The code mentioned here illustrates the different kinds of interpolation method available for `scipy.interpolate.griddata` using 400 points chosen randomly from an interesting function. The results can be compared in Figure 8.16.

Listing 8.16 Interpolation from an unstructured array of two-dimensional points with `scipy.interpolate.griddata`

```
# eg8-gridinterp.py
import numpy as np
from scipy.interpolate import griddata
import matplotlib.pyplot as plt

x = np.linspace(-1,1,100)
y =  np.linspace(-1,1,100)
X, Y = np.meshgrid(x,y)

def f(x, y):
    s = np.hypot(x, y)
    phi = np.arctan2(y, x)
    tau = s + s*(1-s)/5 * np.sin(6*phi)
    return 5*(1-tau) + tau

T = f(X, Y)
# Choose npts random point from the discrete domain of our model function
npts = 400
px, py = np.random.choice(x, npts), np.random.choice(y, npts)

fig, ax = plt.subplots(nrows=2, ncols=2)
# Plot the model function and the randomly selected sample points
ax[0,0].contourf(X, Y, T)
ax[0,0].scatter(px, py, c='k', alpha=0.2, marker='.')
ax[0,0].set_title('Sample points on f(X,Y)')
```

```
# Interpolate using three different methods and plot
for i, method in enumerate(('nearest', 'linear', 'cubic')):
    Ti = griddata((px, py), f(px,py), (X, Y), method=method)
    r, c = (i+1) // 2, (i+1) % 2
    ax[r,c].contourf(X, Y, Ti)
    ax[r,c].set_title('method = {}'.format(method))

plt.show()
```

8.4 Optimization, data-fitting and root-finding

The `scipy.optimize` package provides a range of popular algorithms for minimization of multidimensional functions (with or without additional constraints), least-squares data-fitting and multidimensional equation solving (root-finding). This section will give an overview of the more important options available, but it should be borne in mind that the best choice of algorithm will depend on the individual function being analyzed. For an arbitrary function, there is no guarantee that a particular method will converge on the desired minimum (or root, etc.), or that if it does so it will converge quickly. Some algorithms are better suited to certain functions than others, and the more you know about your function the better. SciPy can be configured to issue a warning message when a particular algorithm fails, and this message can usually help to analyze the problem.

Furthermore, the result returned often depends on the initial guess provided to the algorithm – consider a two-dimensional function as a landscape with several valleys separated by steep ridges: an initial guess placed within one valley is likely to lead most algorithms to wander downhill and find the minimum in that valley (even if it isn't the *global* minimum) without climbing the ridges. Similarly, you might expect (but cannot *guarantee*) that most numerical root-finders return the "nearest" root to the initial guess.

8.4.1 Minimization

SciPy's optimization routines *minimize* a function of one or more variables, $f(x_1, x_2, \cdots, x_n)$. To find the *maximum*, one determines the minimum of $-f(x_1, x_2, \cdots, x_n)$.

Some of the minimization algorithms only require the function itself to be evaluated; others require its first derivative with respect to each of the variables in an array known as the *Jacobian*:

$$J(f) = \left(\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, \cdots, \frac{\partial f}{\partial x_n} \right)$$

Some algorithms will attempt to estimate the Jacobian numerically if it cannot be provided as a separate function.

Furthermore, some sophisticated optimization algorithms require information about the second derivatives of the function, a symmetric matrix of values called the *Hessian*:

$$H(f) = \begin{pmatrix} \frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_2 \partial x_1} & \cdots & \frac{\partial^2 f}{\partial x_n \partial x_1} \\ \frac{\partial^2 f}{\partial x_1 \partial x_2} & \frac{\partial^2 f}{\partial x_2^2} & \cdots & \frac{\partial^2 f}{\partial x_n \partial x_2} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_1 \partial x_n} & \frac{\partial^2 f}{\partial x_2 \partial x_n} & \cdots & \frac{\partial^2 f}{\partial x_n^2} \end{pmatrix}$$

Just as the Jacobian represents local *gradient* of a function of several variables, the Hessian represents the local *curvature*.

Unconstrained minimization

The general algorithm for the minimization of multivariate scalar functions is `scipy.optimize.minimize`, which takes two mandatory arguments:

```
minimize(fun, x0, ...)
```

The first is a function object, `fun`, for evaluating the function to be minimized: this function should take an array of values, `x`, defining the point at which it is to be evaluated $(x_1, x_2, \cdots, x_n)$ followed by any further arguments it requires. The second required argument, `x0`, is an array of values representing the initial guess for the minimization algorithm to start at.

In this section we will demonstrate the use of `minimize` with *Himmelblau's function*, a simple two-dimensional function with some awkward features that make it a good test-function for optimization algorithms. Himmelblau's function is

$$f(x, y) = (x^2 + y - 11)^2 + (x + y^2 - 7)^2.$$

The region $-5 \le x \le 5$, $-5 \le y \le 5$ contains one local maximum,

$$f(-0.270845, -0.923039) = 181.617$$

(though the function climbs steeply outside of this region). There are four minima:

$$\begin{aligned} f(3, 2) &= 0, \\ f(-2.805118, 3.131312) &= 0, \\ f(-3.779310, -3.283186) &= 0, \\ f(3.584428, -1.848126) &= 0. \end{aligned}$$

and four saddle points. Figure 8.17 shows a contour plot of the function.

The function may be defined in Python in the usual way:

```
In [x]: def f(X):
   ...:     x, y = X
   ...:     return (x**2 + y - 11)**2 + (x + y**2 - 7)**2
```

where for clarity we have unpacked the array, `X`, holding (x_1, x_2) into the named values $x_1 \equiv x$ and $x_2 \equiv y$.

To find a minimum, call `minimize` with some initial guess, say $(x, y) = (0, 0)$:

```
In [x]: from scipy.optimize import minimize
In [x]: minimize(f, (0,0))
```

Table 8.4 Minimization information dictonary returned by `scipy.optimize.minimize`

Key	Description
`success`	A boolean value indicating whether or not the minimization was successful
`x`	If successful, the solution: the values of $(x_1, x_2, \cdots, x_n)$ at which the function is a minimum. If the algorithm was not successful, `x` indicates the point at which it gave up
`fun`	If successful, the value of the function at the minimum identified as `x`
`message`	A string describing of the outcome of the minimization
`jac`	The value of the Jacobian: if the minimization is successful the values in this array should be close to zero
`hess, hess_inv`	The Hessian and its inverse (if used)
`nfev, njev, nhev`	The number of evaluations of the function, its Jacobian and its Hessian

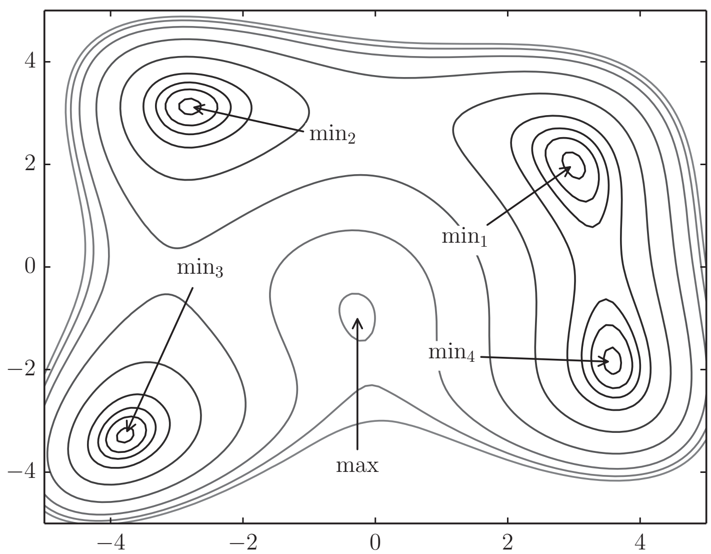


Figure 8.17 Contour plot of Himmelblau's function.

```
      jac: array([ -8.77780211e-06,  -3.52519449e-06])
  message: 'Optimization terminated successfully.'
      fun: 6.15694370233122e-13
     njev: 16
 hess_inv: array([[ 0.01575433, -0.00956965],
       [-0.00956965,  0.03491686]])
   status: 0
     nfev: 64
  success: True
        x: array([ 2.99999989,  1.99999996])
```

`minimize` returns a dictionary-like object with information about the minimization. The important fields are described in Table 8.4: if the minimization is successful, the minimum appears as `x` in this object – here we have converged close to the minimum $f(3,2) = 0$.

Table 8.5 Some of the minimization methods used by `scipy.optimize.minimize`

method	Description
`BFGS`	Broyden-Fletcher-Goldfarb-Shanno (BFGS) algorithm, the default for minimization without constraints or bounds
`Nelder-Mead`	Nelder-Mead algorithm, also known as the downhill simplex or amoeba method. No derivatives are needed
`CG`	Conjugate gradient method
`Powell`	Powell's method (no derivatives are needed with this algorithm)
`dogleg`	Dog-leg trust-region algorithm (unconstrained minimization). Requires the Jacobian and the Hessian (which must be positive-definite)
`TNC`	Truncated Newton algorithm for minimization within bounds
`l-bfgs-b`	Bound-constrained minimization with the L-BFGS-B algorithm
`slsqp`	"Sequential least squares programming" method for minimization with bounds and equality and inequality constraints
`cobyla`	"Constrained optimization by linear approximation" method for constrained minimization

The algorithm to be used by `minimize` is specified by setting its `method` argument to one of the strings given in Table 8.5. The default algorithm, BFGS, is a good general-purpose quasi-Newton method that can approximate the Jacobian if it is not provided and does not use the Hessian. However, it struggles to find the maximum of Himmelblau's function:

```
In [x]: mf = lambda X: -f(X)     # to find the maximum, minimize -f(x,y)
In [x]: minimize(mf, (0,0))
Out[x]:
      jac: array([  1.17853903e+13,   4.57328118e+13])
  message: 'Desired error not necessarily achieved due to precision loss.'
      fun: -2.9978221235736595e+17
     njev: 16
 hess_inv: array([[ 1.03696455, -0.26722678],
       [-0.26722678,  0.0688646 ]])
   status: 2
     nfev: 76
  success: False
        x: array([-14336., -22528.])
```

Starting at $(0, 0)$, the BFGS algorithm has wandered up one of the steep sides of the Himmelblau function (note the size of the Jacobian) and failed to converge. In fact, we need to start quite close to the maximum to succeed:

```
In [x]: minimize(mf, (-0.2,-1))
Out[x]:
      jac: array([  3.81469727e-06,   1.90734863e-06])
  message: 'Optimization terminated successfully.'
      fun: -181.61652152258262
     njev: 8
 hess_inv: array([[ 0.0232834 , -0.00626945],
       [-0.00626945,  0.06137267]])
   status: 0
     nfev: 32
```

```
  success: True
        x: array([-0.27084453, -0.92303852])
```

This is, of course, not much help if we don't know in advance where the maximum is! Let's try a different minimization algorithm, starting at our arbitrary guess, $(0, 0)$:

```
In [x]: minimize(mf, (0,0), method='nelder-mead')
Out[x]:
  status: 0
    nfev: 115
 success: True
 message: 'Optimization terminated successfully.'
     fun: -181.61652150549165
     nit: 59
       x: array([-0.27086815, -0.92300745])
```

The Nelder-Mead algorithm is a simplex method that does not need or estimate the derivatives of the function, so it isn't tempted up the steep sides of the function. However, it has taken 115 function evaluations to converge on the local maximum.

As a final example, consider the `dogleg` method, which requires `minimize` to be passed functions evaluating the Jacobian and the Hessian. The necessary derivatives have simple analytical forms for Himmelblau's function:

$$\frac{\partial f}{\partial x} = 4x(x^2 + y - 11) + 2(x + y^2 - 7)$$

$$\frac{\partial f}{\partial y} = 2(x^2 + y - 11) + 4y(x + y^2 - 7)$$

$$\frac{\partial^2 f}{\partial x^2} = 12x^2 + 4y - 42$$

$$\frac{\partial^2 f}{\partial y^2} = 12y^2 + 4x - 26$$

$$\frac{\partial^2 f}{\partial y \partial x} = \frac{\partial^2 f}{\partial x \partial y} = 4x + 4y$$

The Jacobian and Hessian can be coded up as follows:

```
In [x]: def df:
   ...:     x, y = X
   ...:     f1, f2  = x**2 + y - 11, x + y**2 - 7
   ...:     dfdx = 4*x*f1 + 2*f2
   ...:     dfdy = 2*f1 + 4*y*f2
   ...:     return np.array([dfdx, dfdy])
   ...:
In [x]: def ddf:
   ...:     x, y = X
   ...:     d2fdx2 = 12*x**2 + 4*y - 42
   ...:     d2fdy2 = 12*y**2 + 4*x - 26
   ...:     d2fdxdy = 4*(x + y)
   ...:     return np.array([[d2fdx2, d2fdxdy], [d2fdxdy, d2fdy2]])
   ...:
❶ In [x]: mdf = lambda X: -df(X)
In [x]: mddf = lambda X: -ddf(X)
```

❶ Note that as with the function itself, we need to use the negative of the Jacobian and Hessian if we seek the maximum: these are defined as `lambda` functions `mdf` and `mddf`.

```
In [x]: minimize(mf, (0,0), jac=mdf, hess=mddf, method='dogleg')
Out[x]:
     jac: array([ -1.26922473e-10,   1.23685240e-09])
 message: 'Optimization terminated successfully.'
     fun: -181.6165215225827
    hess: array([[ 44.81187272,   4.77553259],
       [  4.77553259,  16.85937624]])
     nit: 4
    njev: 5
       x: array([-0.27084459, -0.92303856])
  status: 0
    nfev: 5
 success: True
    nhev: 4
```

The algorithm has converged successfully on the local maximum in five function evaluations, five Jacobian evaluations and four Hessian evaluations.

◊ Constrained optimization

Sometimes it is necessary to find the maximum or minimum of a function subject to one or more constraints. To use the earlier mentioned function as an example, you may wish for the single minimum of $f(x, y)$ that satisfies $x > 0$, $y > 0$; or the minimum value of the function along the line $x = y$.

The algorithms `l-bfgs-b`, `tnc` and `slsqp` support the `bounds` argument to `minimize`. `bounds` is a sequence of tuples, each giving the `(min, max)` pairs for each variable of the function defining the bounds on that variable to the minimization. If there is no bound in either direction, use `None`.

For example, if we try to find a minimum in $f(x, y)$ starting at $(-\frac{1}{2}, -\frac{1}{2})$ without specifying any bounds, the `slsqp` method converges (just about) on the one at $(-2.805118, 3.131312)$:

```
In [x]: minimize(f, (-0.5,-0.5), method='slsqp')
Out[x]:
     jac: array([-0.00721077,  0.00037714,  0.        ])
 message: 'Optimization terminated successfully.'
     fun: 4.0198760213901536e-07
     nit: 10
    njev: 10
       x: array([-2.80522924,  3.131319  ])
  status: 0
    nfev: 46
 success: True
```

To stay in the quadrant $x < 0$, $y < 0$, set `bounds` with no minimum on x or y and a maximum bound at $x = 0$ and $y = 0$:

```
In [x]: xbounds = (None, 0)
In [x]: ybounds = (None, 0)
In [x]: bounds = (xbounds, ybounds)
In [x]: minimize(f, (-0.5,-0.5), bounds=bounds, method='slsqp')
```

```
Out[x]:
     jac: array([-0.00283595, -0.00034243,  0.        ])
 message: 'Optimization terminated successfully.'
     fun: 4.115667606325133e-08
     nit: 11
    njev: 11
       x: array([-3.77933774, -3.28319868])
  status: 0
    nfev: 50
 success: True
```

Suppose we wish to find the extrema of Himmelblau's function that also satisfy the condition $x = y$ (that is, they lie along the diagonal of Figure 8.17). Two of the minimization methods listed in Table 8.5 allow for constraints, `cobyla` and `slsqp`, so we must use one of these.

Constraints are specified as the argument `constraints` to the `minimize` function as a sequence of dictionaries defining string keys `'type'`: the *type* of constraint and `'fun'`: a callable object implementing the constraint. `'type'` may be `'eq'` or `'ineq'` for a constraint based on an equality (such as $x = y$) or an inequality (e.g., $x > 2y - 1$). *Note that* `cobyla` *does not support equality constraints*.

An equality constraint function should return zero if the constraint function is met; an inequality constraint function should return a non-negative value if the inequality is met.

To find the minima in $f(x, y)$ subject to the constraint $x = y$, we can use the `slsqp` method with an equality constraint function returning $x - y$:

```
In [x]: con = {'type': 'eq', 'fun': lambda X: X[0] - X[1]}
In [x]: minimize(f, (0,0), constraints=con, method='slsqp')
     jac: array([-16.33084416,  16.33130538,   0.        ])
 message: 'Optimization terminated successfully.'
     fun: 8.0000000007160867
     nit: 7
    njev: 7
       x: array([ 2.54138438,  2.54138438])
  status: 0
    nfev: 32
 success: True
```

The method converged on one of the minima (there is another: start at, for e.g., $(-2, -2)$ to find it). What about the maximum?

```
In [x]: minimize(mf, (0,0), constraints=con, method='slsqp')
Out[x]:
     jac: array([ 0.,  0.,  0.])
 message: 'Singular matrix C in LSQ subproblem'
     fun: -3.1826053300603689e+68
     nit: 4
    njev: 4
       x: array([ -1.12315113e+17,  -1.12315113e+17])
  status: 6
    nfev: 16
 success: False
```

That didn't go so well – the algorithm wandered up the side of a valley. A better choice of algorithm here is `cobyla`, but this method doesn't support equality constraints, so we will build one from a pair of inequalities: $x = y$ if both of $x > y$ and $x < y$ are not satisified:

```
In [x]: con1 = {'type': 'ineq', 'fun': lambda X: X[0] - X[1]}
In [x]: con2 = {'type': 'ineq', 'fun': lambda X: X[1] - X[0]}
In [x]: minimize(mf, (0,0), constraints=(con1, con2), method='cobyla')
Out[x]:
  status: 1
    nfev: 34
 success: True
 message: 'Optimization terminated successfully.'
     fun: -179.12499987327624
   maxcv: 0.0
       x: array([-0.49994148, -0.49994148])
```

Here, the constraint function defined in `con1` returns a non-negative value if $x > y$ and that defined in `con2` returns a non-negative value if $x < y$. The only way both can be satisfied is if $x = y$.

Minimizing a function of one variable

If the function to be minimized is *univariate* (i.e., takes only one variable, a scalar), a faster algorithm is provided by `scipy.optimize.minimize_scalar`. To simply return a minimum, this function can be called with `method='brent'`, which implements Brent's method for locating a minimum.

Ideally, one should "bracket" the minimum first by providing values for x, (a, b, c) such that $f(a) > f(b)$ and $f(c) > f(b)$. This can be done with the `bracket` argument which takes the tuple `(a,b,c)`. If this isn't possible or feasible, provide an interval of two values of x on which to start a search for such a bracket (in the downhill direction). If no `bracket` argument is specified, this search is initiated from the interval $(0, 1)$.

Figure 8.18 gives an example polynomial with two minima and a maximum.

With no `bracket`, `minimize_scalar` converges on the minimum at -2.841 for this function:

```
In [x]: Polynomial = np.polynomial.Polynomial
In [x]: from scipy.optimize import minimize_scalar
In [x]: f = Polynomial( (48., 28., -24., -3., 1.))
In [x]: minimize_scalar(f)
Out[x]:
  fun: -91.32163915433344
 nfev: 11
    x: -2.8410443265958261
  nit: 10
```

If we bracket the other minimum by providing values `(a,b,c)=(3, 4, 6)` which can be seen from Figure 8.18 to satisfy $f(a) > f(b) < f(c)$, the algorithm converges on 4.549:

```
In [x]: minimize_scalar(f, bracket=(3,4,6))
Out[x]:
  fun: -175.45563549487974
```

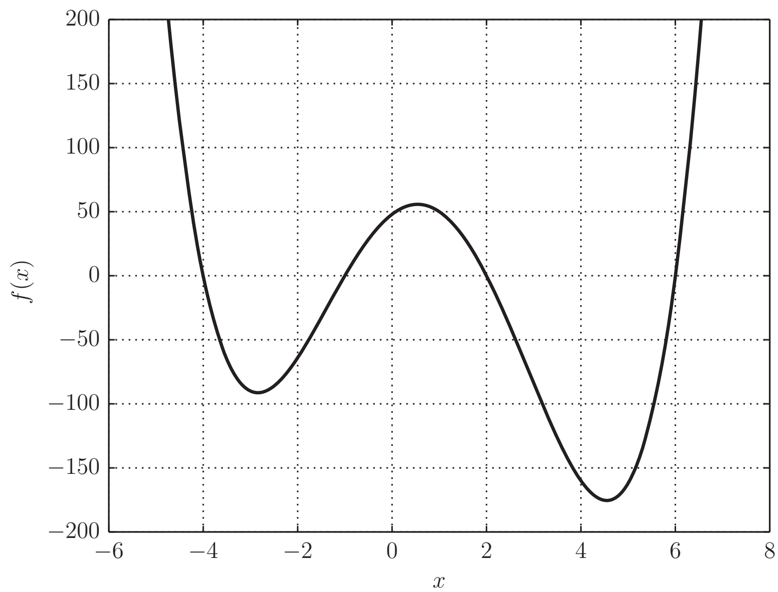


Figure 8.18 The polynomial $f(x) = x^4 - 3x^3 - 24x^2 + 28x + 48$.

```
 nfev: 11
    x: 4.5494683642571934
  nit: 10
```

Finally, to find the maximum, call `minimize_scalar` with $-f(x)$. This time we will initialize a search for a bracket to the minimum of $-f(x)$ with the pair of values $(-1, 0)$:

```
In [x]: minimize_scalar(-f, bracket=(-1, 0))
Out[x]:
  fun: -55.734305899213226
 nfev: 9
    x: 0.54157595897344157
  nit: 8
```

Example E8.21 A simple model for the envelope of an airship treats it as the volume of revolution obtained from a pair of quarter-ellipses joined at their (equal) semi-minor axes. The semi-major axis of the aft ellipse is taken to be longer than that representing the bow by a factor $\alpha = 6$. Equations describing the cross section (in the vertical plane) of the airship envelope may be written

$$y = \begin{cases} \frac{b}{a}\sqrt{x(2a - x)} & (x \leq a), \\ \frac{b}{a}\sqrt{a^2 - \frac{(x-a)^2}{\alpha^2}} & (a < x \leq \alpha(a+1)). \end{cases}$$

The drag on the envelope is given by the formula

$$D = \tfrac{1}{2}\rho_{\mathrm{air}} v^2 V^{2/3} C_{\mathrm{DV}},$$

where ρ_{air} is the air density, v the speed of the airship, V the envelope volume and the drag coefficient, C_{DV} is estimated using the following empirical formula:[10]

$$C_{\text{DV}} = \text{Re}^{-1/6}[0.172(l/d)^{1/3} + 0.252(d/l)^{1.2} + 1.032(d/l)^{2.7}].$$

Here, $\text{Re} = \rho_{\text{air}} v l / \mu$ is the Reynold's number and μ the dynamic viscosity of the air. l and d are the airship length and maximum diameter ($= 2b$) respectively.

Suppose we want to minimize the drag with respect to the parameters a and b but fix the total volume of the airship envelope, $V = \frac{2}{3}\pi a b^2 (1 + \alpha)$. The following program does this using the `slsqp` algorithm, for a volume of 200000 m^3, that of the Hindenburg.

Listing 8.17 Minimizing the drag on an airship envelope

```
# eg8-airship.py
import numpy as np
from scipy.optimize import minimize

# air density (kg.m-3) and dynamic viscosity (Pa.s) at cruise altitude
rho, mu = 1.1, 1.5e-5
# air speed (m.s-1) at cruise altitude
v = 30

def CDV(L, d):
    """ Calculate the drag coefficient. """
    Re = rho * v * L / mu       # Reynold's number
    r = L / d                   # "Fineness" ratio
    return (0.172 * r**(1/3) + 0.252 / r**1.2 + 1.032 / r**2.7) / Re**(1/6)

def D(X):
    """ Return the total drag on the airship envelope. """
    a, b = X
    L = a * (1+alpha)
    return 0.5 * rho * v**2 * V(X)**(2/3) * CDV(L, 2*b)

# Fixed total volume of the airship envelope (m3)
V0 = 2.e5
# Parameter describing the tapering of the stern of the envelope
alpha = 6

def V(X):
    """ Return the volume of the envelope. """
    a, b = X
    return 2 * np.pi * a * b**2 * (1+alpha) / 3

# Minimize the drag, constraining the volume to be equal to V0
a0, b0 = 70, 45     # initial guesses for a, b
con = {'type': 'eq', 'fun': lambda X: V(X)-V0}
res = minimize(D, (a0, b0), method='slsqp', constraints=con)
if res['success']:
    a, b = res['x']
    L, d = a * (1+alpha), 2*b       # length, greatest diameter
```

[10] S. F. Hoerner, *Fluid Dynamic Drag*, Hoerner Fluid Dynamics (1965).

```
    print('Optimum parameters: a = {:g} m, b = {:g} m'.format(a, b))
    print('V = {:g} m3'.format(V(res['x'])))
    print('Drag, D = {:g} N'.format(res['fun']))
    print('Total length, L = {:g} m'.format(L))
    print('Greatest diameter, d = {:g} m'.format(d))
    print('Fineness ratio, L/d = {:g}'.format(L/d))
else:
    # We failed to converge: output the results dictionary
    print('Failed to minimize D!', res, sep='\n')
```

This example is a little contrived, since for fixed α the requirement that V be constant means that a and b are not independent, but a solution is found readily enough:

```
Optimum parameters: a = 32.9301 m, b = 20.3536 m
V = 200000 m3
Drag, D = 20837.6 N
Total length, L = 230.51 m
Greatest diameter, d = 40.7071 m
Fineness ratio, L/d = 5.66266
```

The actual dimensions of the Hindenburg were $l = 245$ m, $d = 41$ m giving the ratio $l/d = 5.98$; so we didn't do too badly.

8.4.2 Nonlinear least squares fitting

SciPy's general *nonlinear* least squares fitting routine is `scipy.optimize.leastsq`, which has the most basic call signature:

`scipy.optimize.leastsq(func, x0, args=())`.

This will attempt to fit a sequence of data points, `y`, to a model function, `f`, which depends on one or more fit parameters. `leastsq` is passed a related function object, `func`, which returns the *difference* between `y` and `f` (the *residuals*). `leastsq` also requires an initial guess for the fitted parameters, `x0`. If `func` requires any other arguments (typically, arrays of the data, `y`, and one or more independent variables), pass them in the sequence `args`. For example, consider fitting the artificial noisy decaying cosine function, $f(t) = Ae^{t/\tau} \cos 2\pi \nu t$ (Figure 8.19).

```
In [x]: import numpy as np
In [x]: import pylab

In [x]: A, freq, tau = 10, 4, 0.5
In [x]: def f(t, A, freq, tau):
   ...:     return A * np.exp(-t/tau) * np.cos(2*np.pi * freq * t)
   ...:
In [x]: tmax, dt = 1, 0.01
In [x]: t = np.arange(0, tmax, dt)
In [x]: yexact = f(t, A, freq, tau)
In [x]: y = yexact + np.random.randn(len(yexact))*2
In [x]: pylab.plot(t, yexact)
In [x]: pylab.plot(t, y)
In [x]: pylab.show()
```

To fit this noisy data, `y`, to the parameters `A`, `freq` and `tau` (pretending we don't know them), we first define our `residuals` function:

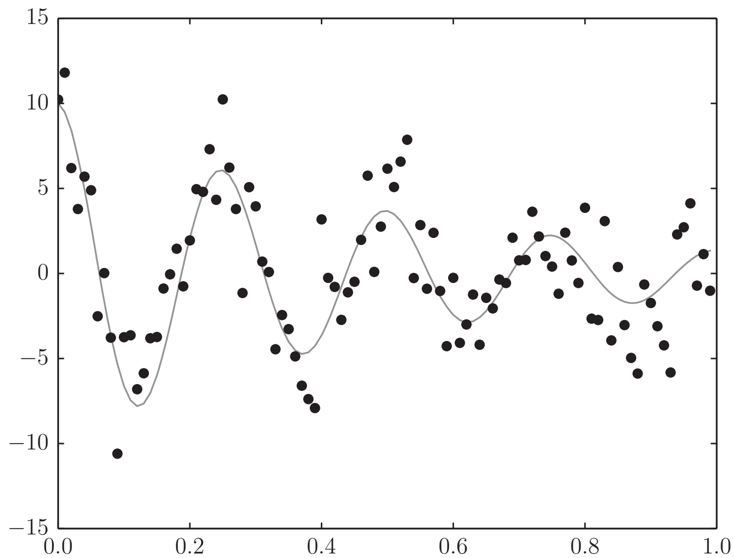


Figure 8.19 A synthetic noisy decaying cosine function.

```
In [x]: def residuals(p, y, t):
   ...:     A, freq, tau = p
   ...:     return y - f(t, A, freq, tau)
```

The first argument is the sequence of parameters, `p`, which we unpack into named variables for clarity. The additional arguments needed are the data itself, `y`, and the independent variable, `t`. Now make some initial guesses for the parameters that aren't too wildly off and call `leastsq`:

```
In [x]: from scipy.optimize import leastsq
In [x]: p0 = 5, 5, 1
In [x]: plsq = leastsq(residuals, p0, args=(y, t))
In [x]: plsq[0]
Out[x]: [ 9.33962672  4.04958427  0.48637434]
```

As with SciPy's other optimization routines, `leastsq` can be configured to return more information about its working, but here we report only the solution (best fit parameters), which is always the first item in the `plsq` tuple.

The true values are `A, freq, tau = 10, 4, 0.5`, so given the noise we haven't done badly. Graphically,

```
In [x]: pylab.plot(t, y, 'o', c='k', label='Data')
In [x]: pylab.plot(t,yexact,c='gray', label='Exact')
In [x]: pylab.plot(t,f(t, *pfit),c='k', label='Fit')
In [x]: pylab.legend()
In [x]: pylab.show()
```

The fit is illustrated in Figure 8.20.

If it is known, it is also possible to pass the Jacobian to `leastsq`, as the following example demonstrates.

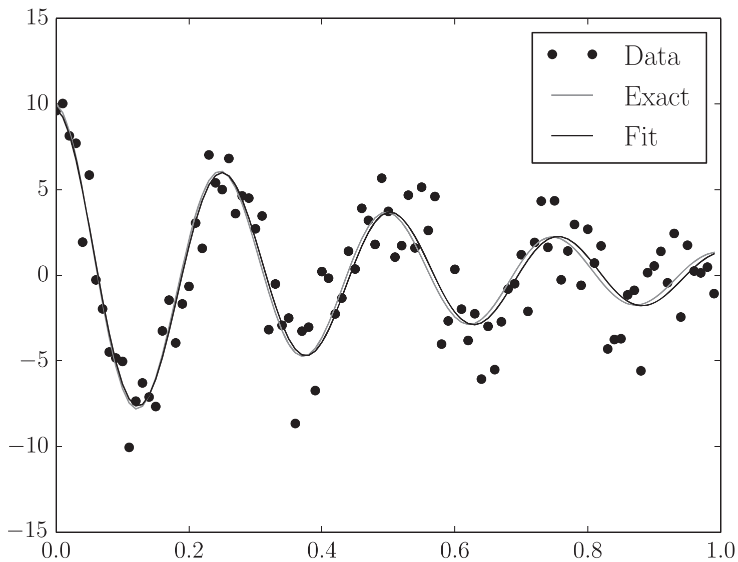


Figure 8.20.

Example E8.22 In this example, we are given a noisy series of data points that we want to fit to an ellipse. The equation for an ellipse may be written as a nonlinear function of angle, θ ($0 \le \theta < 2\pi$), which depends on the parameters a (the semi-major axis) and e (the eccentricity):

$$r(\theta; a, e) = \frac{a(1-e^2)}{1 - e\cos\theta}.$$

To fit a sequence of data points (θ, r) to this function, we first code it as a Python function taking two arguments: the independent variable, `theta`, and a tuple of the parameters, `p = (a, e)`. The function we wish to minimize is the difference between this model function and the data, `r`, defined as the method `residuals`:

```
def f(theta, p):
    a, e = p
    return a * (1 - e**2)/(1 - e*np.cos(theta))

def residuals(p, r, theta):
    return r - f(theta, p)
```

We also need to give `leastsq` an initial guess for the fit parameters, say `p0 = (1,0.5)`. The simplest call to fit the function would then pass to `leastsq` the objects `residuals`, `p0` and `args=(r, theta)` (the additional arguments needed by the `residuals` function:

```
plsq = leastsq(residuals, p0, args=(r, theta))
```

If at all possible, however, it is better to also provide the Jacobian (the first derivative of the fit function with respect to the parameters to be fitted). Expressions for these are straightforward to calculate and implement:

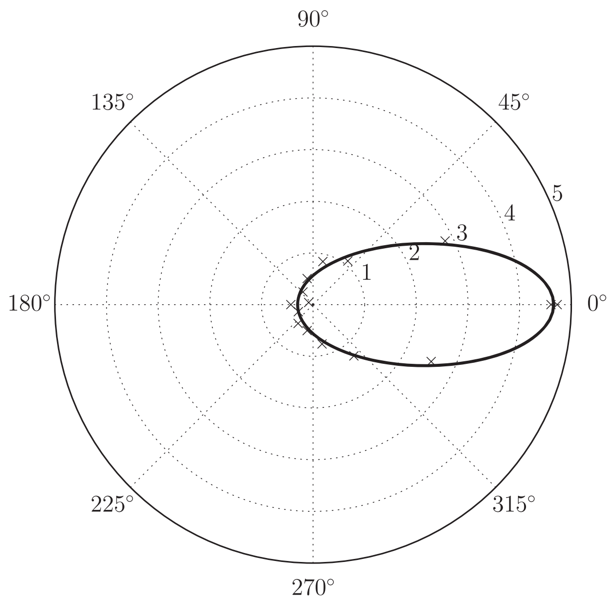


Figure 8.21 Nonlinear least squares fitting of data to the equation of an ellipse in polar coordinates.

$$\frac{\partial f}{\partial a} = \frac{(1-e^2)}{1-e\cos\theta},$$
$$\frac{\partial f}{\partial e} = \frac{a(1-e^2)\cos\theta - 2ae(1-e\cos\theta)}{(1-e\cos\theta)^2}.$$

However, the function we wish to minimize is the residuals function, $r-f$, so we need the negatives of these derivatives. Here is the working code and the fit result (Figure 8.21).

Listing 8.18 Nonlinear least squares fit to an ellipse

```
# eg8-leastsq.py

import numpy as np
from scipy import optimize
import pylab

def f(theta, p):
    a, e = p
    return a * (1 - e**2)/(1 - e*np.cos(theta))

# The data to fit
theta = np.array([0.0000,0.4488,0.8976,1.3464,1.7952,2.2440,2.6928,
                  3.1416,3.5904,4.0392,4.4880,4.9368,5.3856,5.8344,6.2832])
r = np.array([4.6073, 2.8383, 1.0795, 0.8545, 0.5177, 0.3130, 0.0945, 0.4303,
              0.3165, 0.4654, 0.5159, 0.7807, 1.2683, 2.5384, 4.7271])

def residuals(p, r, theta):
    """ Return the observed - calculated residuals using f(theta, p). """
    return r - f(theta, p)
```

```
def jac(p, r, theta):
    """ Calculate and return the Jacobian of residuals. """
    a, e = p
    da = (1 - e**2)/(1 - e*np.cos(theta))
    de = (-2*a*e*(1-e*np.cos(theta)) + a*(1-e**2)*np.cos(theta))/(1 -
                                                   e*np.cos(theta))**2
    return -da,  -de
    return np.array((-da, -de)).T

# Initial guesses for a, e
p0 = (1, 0.5)
plsq = optimize.leastsq(residuals, p0, Dfun=jac, args=(r, theta), col_deriv=True)
print(plsq)

pylab.polar(theta, r, 'x')
theta_grid = np.linspace(0, 2*np.pi, 200)
pylab.polar(theta_grid, f(theta_grid, plsq[0]), lw=2)
pylab.show()
```

SciPy also includes a curve-fitting function, `scipy.optimize.curve_fit`, that can fit data to a function directly (without the need for an additional function to calculate the residuals) and supports weighted least squares fitting. The call signature is

```
curve_fit(f, xdata, ydata, p0, sigma, absolute_sigma)
```

where `f` is the function to fit to the data `(xdata, ydata)`. `p0` is the initial guess for the parameters, and `sigma`, if provided, give the weights of the `ydata` values. If `absolute_sigma` is `True`, these are treated as one standard deviation error (that is, *absolute* weights); the default, `absolute_sigma=False`, treats them as *relative* weights.

The `curve_fit` function returns `popt`, the best-fit values of the parameters and `pcov`, the covariance matrix of the parameters.

Example E8.23 To illustrate the use of `curve_fit` in weighted and unweighted least squares fitting, the following program fits the Lorentzian line shape function centered at x_0 with half width at half-maximum (HWHM), γ, amplitude, A:

$$f(x) = \frac{A\gamma^2}{\gamma^2 + (x - x_0)^2},$$

to some artificial noisy data. The fit parameters are A, γ and x_0. The noise is such that a region of the data close to the line center is much noisier than the rest.

Listing 8.19 Weighted and unweighted least squares fitting with `curve_fit`

```
# eg8-curve-fit.py
import numpy as np
from scipy.optimize import curve_fit
import pylab

x0, A, gamma = 12, 3, 5
```

```
n = 200
x = np.linspace(1, 20, n)
yexact = A * gamma**2 / (gamma**2 + (x-x0)**2)

# Add some noise with a sigma of 0.5 apart from a particularly noisy region
# near x0 where sigma is 3
sigma = np.ones(n)*0.5
sigma[np.abs(x-x0+1)<1] = 3
noise = np.random.randn(n) * sigma
y = yexact + noise

def f(x, x0, A, gamma):
    """ The Lorentzian entered at x0 with amplitude A and HWHM gamma. """
    return A *gamma**2 / (gamma**2 + (x-x0)**2)

def rms(y, yfit):
    return np.sqrt(np.sum((y-yfit)**2))

# Unweighted fit
p0 = 10, 4, 2
popt, pcov = curve_fit(f, x, y, p0)
yfit = f(x, *popt)
print('Unweighted fit parameters:', popt)
print('Covariance matrix:'); print(pcov)
print('rms error in fit:', rms(yexact, yfit))
print()

# Weighted fit
popt2, pcov2 = curve_fit(f, x, y, p0, sigma=sigma, absolute_sigma=True)
yfit2 = f(x, *popt2)
print('Weighted fit parameters:', popt2)
print('Covariance matrix:'); print(pcov2)
print('rms error in fit:', rms(yexact, yfit2))

pylab.plot(x, yexact, label='Exact')
pylab.plot(x, y, 'o', label='Noisy data')
pylab.plot(x, yfit, label='Unweighted fit')
pylab.plot(x, yfit2, label='Weighted fit')
pylab.ylim(-1,4)
pylab.legend(loc='lower center')
pylab.show()
```

As Figure 8.22 shows, the unweighted fit is thrown off by the noisy region. Data in this region are given a lower weight in the weighted fit and so the parameters are closer to their true values and the fit better. The output is

```
Unweighted fit parameters: [ 11.61282984   3.64158981   3.93175714]
Covariance matrix:
[[ 0.0686249  -0.00063262  0.00231442]
 [-0.00063262  0.06031262 -0.07116127]
 [ 0.00231442 -0.07116127  0.16527925]]
rms error in fit: 4.10434012348

Weighted fit parameters: [ 11.90782988   3.0154818    4.7861561 ]
Covariance matrix:
```

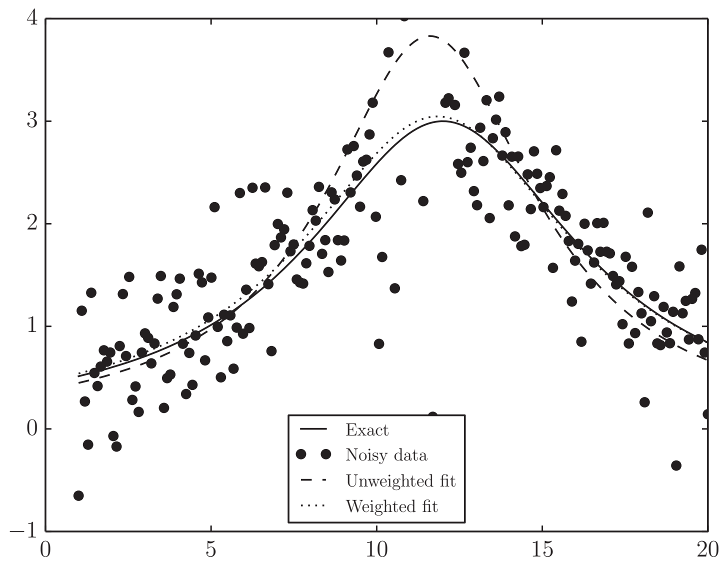


Figure 8.22 Example of least squares fit with `scipy.optimize.curve_fit`.

```
[[ 0.01893474 -0.00333361  0.00639714]
 [-0.00333361  0.01233797 -0.02183039]
 [ 0.00639714 -0.02183039  0.06062533]]
rms error in fit: 0.694013741786
```

8.4.3 Root-finding

`scipy.optimize` provides several methods for obtaining the roots of both univariate and multivariate functions. We describe here only the algorithms relating to functions of a single variable: `brentq`, `brenth`, `ridder` and `bisect`. Each of these methods requires a continuous function, $f(x)$, and a pair of numbers defining a *bracketing interval* for the root to find; that is, values a and b such that the root lies in the interval $[a, b]$ and $f(a) = -f(b)$. Details of the algorithms behind these root-finding methods can be found in standard textbooks on numerical analysis.[11]

In general, the method of choice for finding the root of a well-behaved function is `scipy.optimize.brentq`, which implements a version of Brent's method with inverse quadratic extrapolation (`scipy.optimize.brenth` is a similar algorithm but with hyperbolic extrapolation). As an example, consider the following function for $-1 \le x \le 1$:

$$f(x) = \frac{1}{5} + x\cos\left(\frac{3}{x}\right).$$

[11] For example, Press et al., (2007). *Numerical Recipes*, 3rd ed., Cambridge University Press.

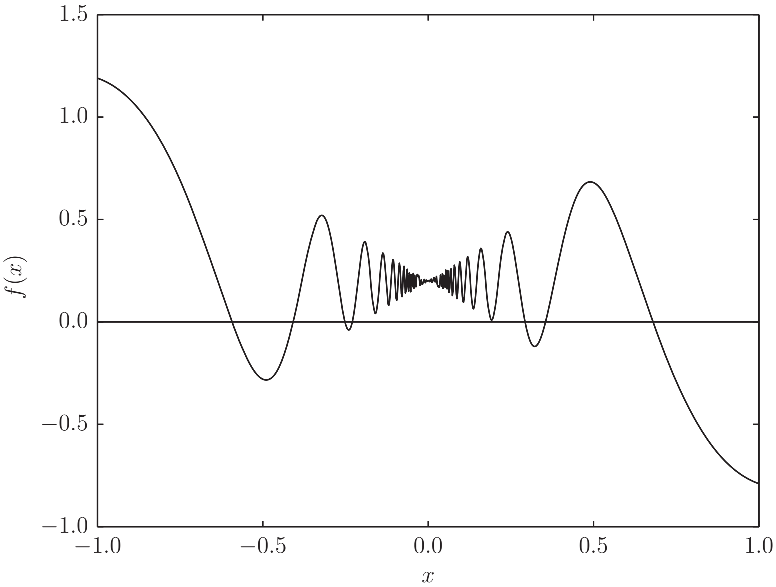


Figure 8.23 The function $f(x) = \frac{1}{5} + x\sin(3/x)$ and its roots.

A plot of this function (Figure 8.23) suggests there is a root between -0.7 and -0.5:

```
In [x]: f = lambda x: 0.2 + x*np.cos(3/x)
In [x]: x = np.linspace(-1, 1, 1000)
In [x]: pylab.plot(x,f(x))
In [x]: pylab.axhline(0, color='k')
In [x]: pylab.show()

In [x]: from scipy.optimize import brentq
In [x]: brentq(f, -0.7, -0.5)
Out[x]: -0.5933306271014237
```

The algorithm for root-finding known as *Ridder's method* is implemented in the function `scipy.optimize.ridder` and the slower but very reliable (for continuous functions) method of bisection is `scipy.optimize.bisect`.

Finally, root-finding by the Newton-Raphson algorithm can be very fast (quadratic) for many continuous functions, provided the first derivative, $f'(x)$, can be calculated. For functions for which an analytical expression for $f'(x)$ can be coded, this is passed to the method `scipy.optimize.newton` as the argument `fprime` along with a starting point, `x0`, which should (in general) be as near to the root as possible. It is not necessary to bracket the root. If the $f'(x)$ cannot be provided, the secant method is used by `newton`. If you are in the happy position of being able to provide the second derivative, $f''(x)$, as `fprime2` as well as the first, Halley's method (which converges even faster than the basic Newton-Raphson algorithm) is used instead.

Note that the stopping condition within the iterative algorithm used by `newton` is the step size so there is no guarantee that it has converged on the desired root: the result should be verified by evaluating the function at the returned value to check that it is (close to) zero.

Table 8.6 Population data for voles measured by Leslie and Ranson

x /weeks	$m(x)$	$P(x)$
8	0.6504	0.83349
16	2.3939	0.73132
24	2.9727	0.58809
32	2.4662	0.43343
40	1.7043	0.29277
48	1.0815	0.18126
56	0.6683	0.10285
64	0.4286	0.05348
72	0.3000	0.02549

Example E8.24 In ecology, the *Euler-Lotka equation* describes the growth of a population in terms of $P(x)$, the fraction of individuals alive at age x and $m(x)$, the mean number of live females born per time period per female alive during that time period:

$$\sum_{x=\alpha}^{\beta} P(x)m(x)e^{-rx} = 1,$$

where α and β are the boundary ages for reproduction defining the discrete growth rate, $\lambda = e^r$. $r = \ln\lambda$ is known as *Lotka's intrinsic rate of natural increase*.

In a paper by Leslie and Ranson,[12] $P(x)$ and $m(x)$ were measured for a population of voles (*Microtus agrestis*) using a time period of eight weeks. The data are given in Table 8.6.

The sum $R_0 = \sum_{x=\alpha}^{\beta} P(x)m(x)$ gives the ratio between the total number of female births in successive generations; a population grows if $R_0 > 1$ and r determines how fast this growth is. In order to find r, Leslie and Ranson used an approximate numerical method; the code mentioned here determines r by finding the real root of the Lotka-Euler equation directly (it can be shown that there is only one).

Listing 8.20 Solution of the Euler-Lotka equation

```
# eg8-euler-lotka.py
import numpy as np
from scipy.optimize import brentq

# The data, from Table 6 of:
# P. H. Leslie and R. M. Ranson, J. Anim. Ecol. 9, 27 (1940)
x = np.linspace(8, 72, 9)
m = np.array( [0.6504, 2.3939, 2.9727, 2.4662, 1.7043,
               1.0815, 0.6683, 0.4286, 0.3000] )
P = np.array( [0.83349, 0.73132, 0.58809, 0.43343, 0.29277,
               0.18126, 0.10285, 0.05348, 0.02549] )
```

[12] P. H. Leslie and R. M. Ranson, (1940). *J. Anim. Ecol.* **9**, 27.

```
# Calculate the product sequence f and R0, the ratio between the number of
# female births in successive generations.
f = P * m
R0 = np.sum(f)
if R0 > 1:
    msg = 'R0 > 1: population grows'
else:
    msg = 'Population does not grow'

# The Euler-Lotka equation: we seek the one real root in r
def func(r):
    return np.sum(f * np.exp(-r * x)) - 1

# Bracket the root and solve with scipy.optimize.brentq
a, b = 0, 10
r = brentq(func, a, b)
print('R0 = {:.3f} ({})'.format(R0, msg))
print('r = {:.5f} (lambda = {:.5f})'.format(r, np.exp(r)))
```

The output of this program is as follows:

```
R0 = 5.904 (R0 > 1: population grows)
r = 0.08742 (lambda = 1.09135)
```

This value of r may be compared with the approximate value obtained by Leslie and Ranson, who comment:

> The required root is 0.087703 which slightly overestimates the value of r, to which the series is approaching. This lies between 0.0861 (the third degree approximation) and 0.0877, but nearer the latter than the former, the error being probably in the last decimal place.

8.4.4 Exercises

Questions

Q8.4.1 Use `scipy.optimize.brentq` to find the solutions to the equation

$$x + 1 = -\frac{1}{(x-3)^3}$$

Q8.4.2 Using `scipy.optimize.newton` to find a root of the following functions (with the given starting point, x_0) fails. Explain why and find the roots either by modifying the call to `newton` or by using a different method.

a.

$$f(x) = x^3 - 5x, \quad x_0 = 1$$

b.

$$f(x) = x^3 - 3x + 1, \quad x_0 = 1$$

c.

$$f(x) = 2 - x^5, \quad x_0 = 0.01$$

d.

$$f(x) = x^4 - (4.29)x^2 - 5.29, \quad x_0 = 0.8$$

Q8.4.3 The trajectory of a projectile in the xz-plane launched from the origin at an angle θ_0 with speed $v_0 = 25\ \mathrm{m\,s^{-1}}$ is

$$z = x\tan\theta_0 - \frac{g}{2v_0^2\cos\theta_0}x^2.$$

If the projectile passes through the point $(5, 15)$, use Brent's method to determine the possible values of θ_0.

Problems

P8.4.1 A rectangular field with area $A = 10,000\ \mathrm{m}^3$ is to be fenced-off beside a straight river (the boundary with the river does not need to be fenced). What dimensions a, b minimize the amount of fencing required? Verify that a constrained minimization algorithm gives the same answer as the algebraic analysis.

P8.4.2 Find all of the roots of

$$f(x) = \frac{1}{5} + x\cos\left(\frac{3}{x}\right)$$

using (a) `scipy.optimize.brentq` and (b) `scipy.optimize.newton`.

P8.4.3 The *Wien displacement law* predicts that the wavelength of maximum emission from a black body described by Planck's law is proportional to $1/T$:

$$\lambda_{\mathrm{max}}T = b,$$

where b is a constant known as *Wien's displacement constant*. Given the Planck distribution of emitted energy density as a function of wavelength,

$$u(\lambda, T) = \frac{8\pi^2 hc}{\lambda^5}\frac{1}{e^{hc/\lambda k_{\mathrm{B}}T} - 1},$$

determine the constant b by using `scipy.optimize.minimize_scalar` to find the maximum in $u(\lambda, T)$ for temperatures in the range $500\ \mathrm{K} \leq T \leq 6000\ \mathrm{K}$ and fitting λ_{max} to a straight line against $1/T$. Compare with the "exact" value of b, which is available within `scipy.constants` (see Section 8.1.1).

P8.4.4 Consider a one-dimensional quantum mechanical particle in a box ($-1 \leq x \leq 1$) described by the Schrödinger equation:

$$-\frac{\mathrm{d}^2\psi}{\mathrm{d}x^2} = E\psi,$$

in energy units for which $\hbar^2/(2m) = 1$ with m the mass of the particle. The exact solution for the ground state of this system is given by

$$\psi = \cos\left(\frac{\pi x}{2}\right), \quad E = \frac{\pi^2}{4}.$$

An approximate solution may be arrived at using the *variational principle* by minimizing the expectation value of the energy of a trial wavefunction,

$$\psi_{\mathrm{trial}} = \sum_{n=0}^{N} a_n \phi_n(x)$$

with respect to the coefficients a_n. Taking the basis functions to have the following symmetrized polynomial form,

$$\phi_n = (1-x)^{N-n+1}(x+1)^{n+1},$$

use `scipy.optimize.minimize` and `scipy.integrate.quad` to find the optimum value of the expectation value (Rayleigh-Ritz ratio):

$$\mathcal{E} = \frac{\langle \psi_{\mathrm{trial}} | \hat{H} | \psi_{\mathrm{trial}} \rangle}{\langle \psi_{\mathrm{trial}} | \psi_{\mathrm{trial}} \rangle} = -\frac{\int_{-1}^{1} \psi_{\mathrm{trial}} \frac{\mathrm{d}^2}{\mathrm{d}x^2} \psi_{\mathrm{trial}} \, \mathrm{d}x}{\int_{-1}^{1} \psi_{\mathrm{trial}} \psi_{\mathrm{trial}} \, \mathrm{d}x}.$$

Compare the estimated energy, $\mathcal{E}$, with the exact answer for $N = 1, 2, 3, 4$. (*Hint:* use `np.polynomial.Polynomial` objects to represent the basis and trial wavefunctions.)

9 General scientific programming

9.1 Floating point arithmetic

9.1.1 The representation of real numbers

The real numbers, such as 1.2, -0.36, π, 4 and 13256.625 may be thought of as points on a continuous, infinite *number line*.[1] Some real numbers (including the integers themselves) can be expressed as a ratio of two integers, for example, $\frac{5}{8}$ and $\frac{1}{3}$. Such numbers are called *rational*. Others, such as π, e and $\sqrt{2}$ cannot and are called *irrational*.

There can therefore be several ways of writing a real number, depending on which category it falls into, and not all of these ways can express the number precisely (using a finite amount of ink!). For example, the rational real number $\frac{5}{8}$ may be written exactly as a *decimal expansion* as 0.625:

$$\frac{5}{8} = \frac{6}{10} + \frac{2}{100} + \frac{5}{1000},$$

but the number $\frac{1}{3}$ cannot be written in a finite number of terms of a decimal expansion:

$$\frac{1}{3} = \frac{3}{10} + \frac{3}{100} + \frac{3}{1000} + \cdots = 0.333\cdots$$

In writing $\frac{1}{3}$ as a decimal expansion we must truncate the infinite sequence of 3s somewhere.

The irrational numbers can be *described* exactly (given some presumed geometrical or other knowledge), for example, π is the ratio of a circle's circumference to its diameter, $\sqrt{2}$ is the length of the hypotenuse of a right-angled triangle whose other sides have length 1. To *represent* or store such a number numerically, however, some level of approximation is necessary. For example, $\frac{355}{113}$ is a famous rational approximation to π. A (better) decimal approximation is 3.14159265358979. But, as a decimal expansion,[2] an infinite number of digits are needed to express the value of π precisely, just as an infinite number of 3s are needed in the decimal expansion of $\frac{1}{3}$.

Computers store numbers in binary, and the same considerations that apply to the limits of the decimal representation of a real number apply to its binary representation.

---

[1] Obviously, an integer such as 4 is just a special sort of real number.

[2] Note that a decimal expansion is simply a rational number with a power of 10 in the denominator, $3.14159265358979 = \frac{314159265358979}{100000000000000}$.

For example, $\frac{5}{8}$ has an exact binary representation in a finite number of bits:

$$\frac{5}{8} = \frac{1}{2} + \frac{0}{4} + \frac{1}{8} = 0.101_2$$

but

$$\frac{1}{10} = 0.0001100110011001100110011\cdots_2\,,$$

an infinitely repeating sequence. Only a finite number of these digits can be stored, and the truncated series of bits converted back to decimal is

$$\frac{1}{10} \approx 0.10000000000000009$$

using the so-called *double-precision* standard common to most computer languages on most operating systems. This is the *nearest representable number* to $\frac{1}{10}$.

The format of the double-precision floating point representation of numbers is dictated by the IEEE-754 standard. There are three parts to the representation, stored in a total of 64 bits (8 bytes): the single *sign bit*, an 11-bit *exponent* and a 52-bit *significand* (also called the *mantissa*). This is best demonstrated by an example in decimal: the number 13256.625 can be written in scientific notation as:

$$13256.625 = +1.3256625 \times 10^4$$

and stored with the sign bit corresponding to `+`, a significand equal to `13256625` (where the decimal point is implicitly to be placed after the first digit) and the exponent `4`. This notation is called "floating point" because the decimal point[3] is moved by the number of places indicated by the exponent.

The floating point representation of numbers in binary works in the same way, except that each digit can only be 0 or 1, of course. This allows for a neat trick: when the number's binary point (equivalent to the decimal point in base-10) is shifted so that its significand has no leading zeros, then it will start with 1. Because all significands *normalized* in this way will start with 1, there is no need to store it, and effectively 53 bits of precision can be stored in a 52-bit significand.[4] The omitted bit is sometimes called the *hidden bit*.

In our example, 13256.625 can in fact be represented exactly in binary as

$$13256.625_{10} \equiv 11001111001000.101_2$$

The normalized form of the significand is therefore `11001111001000101` and the exponent is 13, since:

$$11001111001000.101_2 = 1.1001111001000101 \times 2^{13}.$$

Now, as discussed, the first digit of the normalized signifcand will always be 1, so it is omitted and the significand is stored as

```
1001111001000101000000000000000000000000000000000000
```

[3] More generally known as the *radix* point in bases other than base-10.

[4] Note that this trick works only in the binary sysetm.

In order to allow for negative exponents (numbers with magnitudes less than 1), the exponent is stored with a *bias*: 1023 is added to the actual exponent. That is, actual exponents in the range -1022 to $+1023$ are stored as numbers in the range 1 to 2046. In this case, the 11-bit exponent field is $13 + 1023 = 1036$:

```
10000001100
```

Finally, the sign bit is `0`, indicating a positive number. The full, 64-bit floating point representation of 13256.625 (with spaces for clarity) is

```
0 10000001100 1001111001000101000000000000000000000000000000000000
```

and is exact. However, 0.1 is

```
0 01111111011 1001100110011001100110011001100110011001100110011010
```

and is not exact (note the truncation and rounding of the infinitely repeating sequence 0011...) – in decimal, this number is

```
0.1000000000000000055511151231257
```

In general, the 53 bits (including the *hidden bit*) of the significand give about 15 decimal digits of precision ($\log_{10}(2^{53}) = 15.95$). Any calculation resulting in more significant digits is subject to *rounding error*. The upper bound of the relative error due to rounding is called the *machine epsilon*, ϵ. In Python,

```
In [x]: import sys
In [x]: eps = sys.float_info.epsilon
In [x]: eps
Out[x]: 2.220446049250313e-16
```

It can be shown that the maximum spacing between two normalized floating point numbers is 2ϵ. That is, `x + 2*eps == x` is guaranteed always to be `False`.

9.1.2 Comparing floating point numbers

Because of the finite precision of the floating point representation of (most) real numbers it is extremely risky to compare two `floats` for equality. For example, consider squaring 0.1:

```
In [x]: (0.1)**2
Out[x]: 0.010000000000000002
```

As we have come to expect, this is not exactly 0.01, but it is also *not even the nearest representable number to 0.01*, since the number squared was, in fact, 0.10000000000000009. The unfortunate consequence of this is

```
In [x]: (0.1)**2 == 0.01
Out[x]: False
```

NumPy provides the methods `isclose` and `allclose` (see Section 6.1.12) for comparing two floating point numbers or arrays to within a specified or default tolerance:

```
In [x]: np.isclose(0.1**2, 0.01)
Out[x]: True
```

Note also that floating point addition is not necessarily *associative*:

```
In [x]: a, b, c = 1e14, 25.44, 0.74
In [x]: (a + b) + c
Out[x]: 100000000000026.17

In [x]: a + (b + c)
Out[x]: 100000000000026.19
```

Nor, in general, is floating point multiplication *distributive* over addition:

```
In [x]: a, b, c = 100, 0.1, 0.2

In [x]: a*b + a*c
Out[x]: 30.0

In [x]: a * (b + c)
Out[x]: 30.000000000000004
```

9.1.3 Loss of significance

Most floating point operations (such as addition and subtraction) result in a loss of significance. That is, the number of significant digits in the result can be smaller than in the original numbers (operands) used in the calculation. To illustrate this, consider a hypothetical floating point representation working in decimal with a 6-digit significand and perform the following calculation, written in its exact form:

$$1.2345432 - 1.23451 = 0.0000332$$

Our hypothetical system cannot store the first operand to its full precision but can only get as close as 1.23454. The floating point subtraction then yields

$$1.23454 - 1.23451 = 0.00003.$$

The original numbers were accurate in the most significant six digits, but the result is only accurate in its first significant digit. Note that it isn't the case that the exact result cannot be *represented* in all its digits by our floating point architecture: $0.0000332 \equiv 3.32 \times 10^{-5}$ only has three significant digits, well within the six available to us. The drastic loss of significance occurred because there was only a very small difference between the two numbers. This effect is sometimes called *catastrophic cancellation* and should always be a consideration when subtracting two numbers with similar values.

A similar loss of significance can occur when a small number is subtracted from (or added to) a much larger one:

$$\begin{aligned} 12345.6 + 0.123456 &= 12345.72345 \quad &\text{(exactly)},\\ 12345.6 + 0.123456 &= 12345.7 \quad &\text{(6-digit decimal significand)}. \end{aligned}$$

Even though the 15 or so significant digits of a double-precision floating point number may seem like sufficient accuracy for a single calculation, be aware that repeatedly carrying out such calculations can increase this rounding error dramatically if the numbers involved cannot be represented exactly. For example, consider the following:

```
In [x]: for i in range(10000000):
   ....:     a += 0.1
   ....:

In [x]: a
Out[x]: 999999.9998389754
```

The difference between this approximate value and the exact answer, 1000000, is over 1.61×10^{-4}.

Python's `math` module has a function, `fsum`, which uses a technique called the *Shewchuk algorithm* to compensate for rounding errors and loss of significance. Compare these two implementations of the previous sum using a generator expression:

```
In [x]: sum((0.1 for i in range(10000000)))
Out[x]: 999999.9998389754
In [x]: math.fsum((0.1 for i in range(10000000)))
Out[x]: 1000000.0
```

9.1.4 Underflow and overflow

Another consequence of the way that floating point numbers are handled is that there is a minimum and maximum magnitude of number that can be stored. For example, Bayesian calculations frequently require small probabilities to be multiplied together, with each probability a number between 0 and 1. For a large number of such probabilities this product can reach a value that is too small to represent resulting in *underflow* to zero:

```
In [x]: P = 1
In [x]: for i in range(101):
  ....:     print(P)
  ....:     P *= 5.e-4

1
0.0005
2.5e-07
1.25e-10
6.250000000000001e-14
...
1.0097419586828971e-307
5.0487097934146e-311          # denormalization starts
2.5243548965e-314
1.2621776e-317
6.31e-321
5e-324
0.0                           # underflow
0.0
```

Below this value, Python begins to sacrifice some of the precision and maintains a modified representation of the number (a denormal, or subnormal number), a process called *gradual underflow*. Eventually, however, the number underflows its representation totally and becomes indistinguishable from zero. The minimum number that can be represented at full IEEE-754 double precision is

```
In [x]: import sys
In [x]: sys.float_info.min
Out[x]: 2.2250738585072014e-308
```

There are several possible tactics for dealing with underflow (beyond using higher precision numbers such as `np.float128`). In the earlier example, it is common to take the sum of the logarithms of the probabilities, which has a much more modest magnitude, instead of taking the product directly. Alternatively, one could start the earlier code with `P = 1.e100` and manipulate the resulting numbers on the understanding that they are larger than they should be by this constant factor.

Floating point *overflow* is the problem at the other end of the number scale: the largest double-precision number that can be represented is

```
In [x]: sys.float_info.max
Out[x]: 1.7976931348623157e+308
```

In NumPy, numbers that overflow are set to the special values `inf` or `-inf` depending on sign:

```
In [x]: f = 1
In [x]: for x in range(1,40,4):
   ...:     print('exp({}) = {}'.format(x**2, np.exp(x**2)))
   ...:
exp(1) = 2.718281828459045
exp(25) = 72004899337.38588
exp(81) = 1.5060973145850306e+35
exp(169) = 2.487524928317743e+73
exp(289) = 3.2441824460394912e+125
exp(441) = 3.340923407659982e+191
exp(625) = 2.7167594696637367e+271
exp(841) = inf
exp(1089) = inf
exp(1369) = inf
```

This leads to some curious relations between numbers that are too big to represent:

```
In [x]: a, b = 1.e500, 1.e1000
In [x]: a == b
Out[x]: True
In [x]: a, b
Out[x]: (inf, inf)
```

There is another special value, `nan` ("not-a-number", NaN), which is returned by some operations involving overflowed numbers:

```
In [x]: a / b
Out[x]: nan
```

(NumPy also implements its own values, `numpy.nan` and `numpy.inf`, see Section 6.1.4.) Never check if an object is `nan` with the `==` operator: `nan` is not even equal to itself(!)[5]:

[5] Note that this means that the `==` operator is not an *equivalence relation* for floating point numbers as it is not reflexive.

```
In [x]: c = a / b
In [x]: c == c
Out[x]: False
```

Python `int` objects are not subject to overflow, as Python will automatically allocate memory to hold them to full precision (within the limitations of available machine memory). However, NumPy integer arrays, which map to the underlying C data structures are stored in a fixed number of bytes (see Table 6.2) and may overflow. For example,

```
In [x]: a = np.zeros(3, dtype=np.int16)
In [x]: a[:] = -30000, 30000, 40000
In [x]: a
Out[x]: array([-30000,  30000, -25536], dtype=int16)

In [x]: b = np.zeros(3, dtype=np.uint16)
In [x]: b[:] = -30000, 40000, 70000
In [x]: b
Out[x]: array([35536, 40000,  4464], dtype=uint16)
```

Signed 16-bit integers have the range -32768 to 32767 (-2^{15} to $(2^{15}-1)$). Due to the way they are stored, an attempted assignment to the number 40000 has resulted instead in the assignment of $40000-2^{16}=-25536$ to `a[2]` above. Similarly, *unsigned* 16-bit integers are limited to values in the range 0 to 65535 (0 to $(2^{16}-1)$). Negative numbers cannot be represented at all and `b[0] = −30000` gets converted to $-30000 \mod 2^{16}=35536$; `b[2] = 70000` overflows and ends up as $70000 \mod 2^{16}=4464$.

9.1.5 Further Reading

- From the Python documentation: *Floating Point Arithmetic: Issues and Limitations*, available at http://docs.python.org/tutorial/floatingpoint.html.
- The article "What Every Computer Scientist Should Know About Floating-Point Arithmetic" by David Goldberg (*Computing Surveys*, March 1991) has become something of a classic and for a rigorous approach to the topic of floating point arithmetic is highly recommended. It is available at https://docs.oracle.com/cd/E19957-01/806-3568/ncg_goldberg.html.
- S. Oliveira and D. Stewart, (2006).*Writing Scientific Software: A Guide to Good Style*, Cambridge University Press.
- N. J. Higham, (2002). *Accuracy and Stability of Numerical Algorithms*, 2nd ed., Society for Industrial and Applied Mathematics.

9.1.6 Exercises

Questions

Q9.1.1 The *decimal representation* of some real numbers is not unique. For example, prove mathematically that $0.\dot{9} \equiv 0.9999\cdots \equiv 1$.

Q9.1.2 $\sqrt{\tan(\pi)} = 0$ is mathematically well-defined, so why does the folllowing calculation fail with a math domain error?

```
In [x]: math.sqrt(math.tan(math.pi))
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-135-7bfdceeef434> in <module>()
----> 1 math.sqrt(math.tan(math.pi))
```

Q9.1.3 Fermat's Last Theorem states that no three positive integers x, y and z can satisfy the equation $x^n + y^n - z^n = 0$ for any integer $n > 2$. Explain this apparent counter-example to the theorem:

```
In [x]: 844487.**5 + 1288439.**5 - 1318202.**5
Out[x]: 0.0
```

Q9.1.4 The functions $f(x) = (1 - \cos^2 x)/x^2$ and $g(x) = \sin^2 x/x^2$ are mathematically indistinguishable, but plotted using Python in the region $-0.001 \le x \le 0.001$ show a significant difference. Explain the origin of this difference.

Q9.1.5 How can you establish whether a floating point number is `nan` or not without using `math.isnan` or `numpy.isnan`?

Q9.1.6 Predict and explain the outcome of the following:

a. `1e1001 > 1e1000`
b. `1e350/1.e100 == 1e250`
c. `1e250 * 1.e-250 == 1e150 * 1.e-150`
d. `1e350 * 1.e-350 == 1e450 * 1.e-450`
e. `1 / 1e250 == 1e-250`
f. `1 / 1e350 == 1e-350`
g. `1e450/1e350 != 1e450 * 1e-350`
h. `1e250/1e375 == 1e-125`
i. `1e35 / (1e1000 - 1e1000) == 1 / (1e1000 - 1e1000)`
j. `1e1001 > 1e1000 or 1e1001 < 1e1000`
k. `1e1001 > 1e1000 or 1e1001 <= 1e1000`

Problems

P9.1.1 *Heron's formula* for the area of a triangle (as used in Example E2.3)

$$A = \sqrt{s(s-a)(s-b)(s-c)} \text{ where } s = \tfrac{1}{2}(a+b+c)$$

is inaccurate if one side is very much smaller than the other two ("needle-shaped" triangles). Why? Demonstrate that the following reformulation gives a more accurate result in this case by considering the triangle with sides $(10^{-13}, 1, 1)$, which has the area 5×10^{-14}.[6]

$$A = \frac{1}{4}\sqrt{(a + (b + c))(c - (a - b))(c + (a - b))(a + (b - c))},$$

where the sides have been relabeled so that $a \ge b \ge c$.

[6] This formula is due to William Kahan, one of the designers of the IEEE-754 floating point standard.

What happens if you rewrite the factors in this equation to remove their inner parentheses? Why?

P9.1.2 Write a function to determine the machine epsilon of a numerical data type (`float`, `np.float128`, `int`, etc.).

9.2 Stability and conditioning

9.2.1 The stability of an algorithm

The stability of an algorithm may be thought of in relation to how it handles approximation errors that occur in its operation or its input data. These errors typically arise from experimental uncertainties (imperfect measurements providing the input data) or from the sort of floating point approximations involved in the calculations of the algorithm discussed in the previous section. Another common source of error is in the approximations made in "discretizing" a problem: the need to represent the values of a continuous function, $y = f(x)$ say, on a discrete "grid" of points: $y_i = f(x_i)$. An algorithm is said to be numerically stable if it does not magnify these errors and unstable if it causes them to grow.

Example E9.1 Consider the differential equation,

$$\frac{\mathrm{d}y}{\mathrm{d}x} = -\alpha y$$

for $\alpha > 0$ subject to the boundary condition $y(0) = 1$. This simple problem can be solved analytically:

$$y = e^{-\alpha x},$$

but suppose we want to solve it numerically. The simplest approach is the *forward* (or *explicit*) *Euler* method: choose a step size, h, defining a grid of x values, $x_i = x_{i-1} + h$, and approximate the corresponding y values through:

$$y_i = y_{i-1} + h\left|\frac{\mathrm{d}y}{\mathrm{d}x}\right|_{x_{i-1}} = y_{i-1} - h\alpha y_{i-1} = y_{i-1}(1 - \alpha h).$$

The question arises: what value should be chosen for h? A small h minimizes the error introduced by the approximation above which basically joins y values by straight-line segments,[7] but if h is too small there will be cancellation errors due to the finite precision used in representing the numbers involved.[8]

---

[7] That is, the Taylor series about y_{i-1} has been truncated at the linear term in h.

[8] In the extreme case that h is chosen to be smaller than the *machine epsilon*, typically about 2×10^{-16}, then we have $x_i = x_{i-1}$ and there is no grid of points at all.

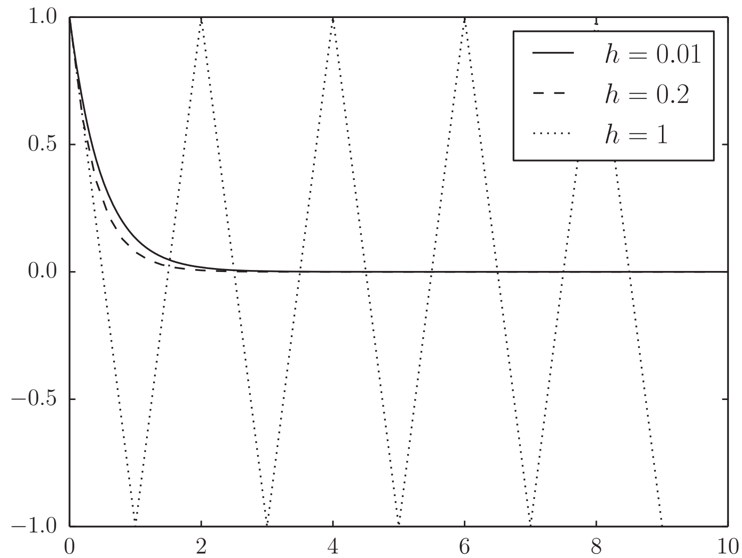


Figure 9.1 Instability of the forward-Euler solution to $\mathrm{d}y/\mathrm{d}x = -\alpha y$ for large step size, h.

The following code implements the forward Euler algorithm to solve the earlier differential equation. The largest value of h (here, $h = \alpha/2 = 1$) clearly makes the algorithm unstable (see Figure 9.1).

Listing 9.1 Comparison of different step sizes, h, in the numerical solution of $y' = -\alpha y$ by the forward Euler algorithm

```
import numpy as np
import pylab

alpha, y0, xmax = 2, 1, 10

def euler_solve(h, n):
    """ Solve dy/dx = -alpha.y by forward Euler method for step size h."""
    y = np.zeros(n)
    y[0] = y0
    for i in range(1, n):
        y[i] = (1 - alpha * h) * y[i-1]
    return y

def plot_solution(h):
    x = np.arange(0, xmax, h)
    y = euler_solve(h, len(x))
    pylab.plot(x, y, label='$h={}$'.format(h))

for h in (0.01, 0.2, 1):
    plot_solution(h)

pylab.legend()
pylab.show()
```

Example E9.2 The integral

$$I_n = \int_0^1 x^n e^x \, \mathrm{d}x \quad n = 0, 1, 2, \cdots$$

suggests a recursion relation obtained by integration by parts:

$$I_n = \left[x^n e^x\right]_0^1 - n \int_0^1 x^{n-1} e^x \, \mathrm{d}x = e - nI_{n-1}$$

terminating with $I_0 = e - 1$. However, this algorithm, applied "forward" for increasing n is numerically unstable since small errors (such as floating point rounding errors) are magnified at each step: if the error in I_n is ϵ_n such that the estimated value of $I'_n = I_n + \epsilon_n$ then

$$\epsilon_n = I'_n - I_n = (e - nI'_{n-1}) - (e - nI_{n-1}) = n(I_{n-1} - I'_{n-1}) = -n\epsilon_{n-1},$$

and hence $|\epsilon_n| = n!\epsilon_0$. Even if the error in ϵ_0 is small, that in ϵ_n is larger by a factor $n!$, which can be huge.

The numerically stable solution, in this case, is to apply the recursion backward for decreasing n:

$$I_{n-1} = \frac{1}{n}(e - I_n) \quad \Rightarrow \epsilon_{n-1} = -\frac{\epsilon_n}{n}.$$

That is, errors in I_n are *reduced* on each step of the recursion. One can even start the algorithm at $I'_N = 0$ and providing enough steps are taken between N and the desired n it will converge on the correct I_n.

Listing 9.2 Comparison of algorithm stability in the calculation of $I(n) = \int_0^1 x^n e^x \, \mathrm{d}x$

```
# eg9-integral-stability.py
import numpy as np
import pylab

def Iforward(n):
    if n == 0:
        return np.e - 1
    return np.e - n * Iforward(n-1)

def Ibackward(n):
    if n >= 99:
        return 0
    return (np.e - Ibackward(n+1)) / (n+1)

N = 35
Iforward = [np.e - 1]
for n in range(1, N+1):
    Iforward.append(np.e - n * Iforward[n-1])

Ibackward = [0] * (N+1)
for n in range(N-1,-1,-1):
    Ibackward[n] = (np.e - Ibackward[n+1]) / (n+1)

n = range(N+1)
pylab.plot(n, Iforward, label='Forward algorithm')
```

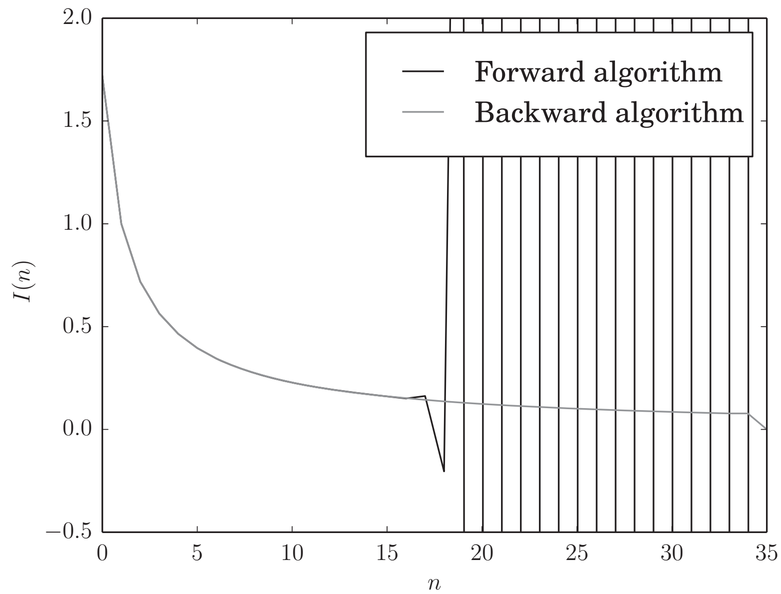


Figure 9.2 Instability of the forward recursion relation for $I_n = \int_0^1 x^n e^x \, dx$.

```
pylab.plot(n, Ibackward, label='Backward algorithm')
pylab.ylim(-0.5, 2)
pylab.xlabel('$n$')
pylab.ylabel('$I(n)$')
pylab.legend()
pylab.show()
```

Figure 9.2 shows the forward algorithm becoming extremely unstable for $n > 16$ and fluctuating between very large positive and negative values; conversely, the backward algorithm is well behaved.

9.2.2 Well-conditioned and ill-conditioned problems

In numerical analysis, a further distinction is made between problems which are well- or ill-conditioned. A *well-conditioned problem* is one for which small relative errors in the input data lead to small relative errors in the solution; an *ill-conditioned problem* is one for which small input errors lead to large errors in the solution. Conditioning is a property of the problem, not the algorithm and is distinct from the issue of stability: it is perfectly possible to use an unstable algorithm on a well-conditioned problem and end up with erroneous results.

Example E9.3 Consider the two lines given by the equations:

$$y = x$$
$$y = mx + c$$

These lines intersect at $(x_\star, y_\star) = (c/(1-m), c/(1-m))$. Finding the intersection point is an ill-conditioned problem when $m \approx 1$ (lines nearly parallel).

For example, the lines $y = x$ and $y = (1.01)x+2$ intersect at $(x_\star, y_\star) = (-200, -200)$. If we perturb m slightly by $\delta m = 0.001$, to $m' = m + \delta m = 1.011$, the intersection point becomes $(x'_\star, y'_\star) = (-181.8182, -181.8182)$. That is, a relative error of $\delta m/m \approx 0.001$ in m has created a relative error of $|(x'_\star - x_\star)/x_\star| \approx 0.091$, almost 100 times larger.

Conversely, if the lines have very different gradients, the problem is well-conditioned. Take, for example, $m = -1$ (perpendicular lines): the intersection $(1, 1)$ becomes $(1.0005, 1.0005)$ under the same perturbation to $m' = m + \delta m = -0.999$, leading to a relative error of 0.0005, which is actually *smaller* than the relative error in m.

Example E9.4 The conditioning of polynomial root-finding is notoriously bad. One famous example is *Wilkinson's polynomial*:

$$\begin{aligned} P(x) &= \prod_{i=1}^{20}(x-i) = (x-1)(x-2)\cdots(x-20) \\ &= x^{20} - 210x^{19} + 20615x^{18} + \cdots + 2432902008176640000 \end{aligned}$$

By inspection, the roots are simply $1, 2, \cdots, 20$. However, Wilkinson showed that decreasing the coefficient of x^{19} from -210 to $-210 - 2^{-23} \approx -210.000000119209$ had a drastic effect on many of the roots, some of which become complex. For example, the root at $x = 20$ moves to $x = 20.8$, a change of 4% on a perturbation of one coefficient by less than one part in a billion (see also Problem 9.2.2).

Problems

P9.2.1 The simplest (and least accurate) way to calculate the first derivative of a function is to simply use the definition:

$$f'(x) = \lim_{h\to 0}\frac{f(x+h) - f(x)}{h}.$$

Fixing h at some small value, our approximation is

$$f'(x) \approx \frac{f(x+h) - f(x)}{h}.$$

Using the function $f(x) = e^x$, which value of h (to the nearest power of 10) gives the most accurate approximation to $f'(1) = e$?

P9.2.2 Use NumPy's `Polynomial` class (see Section 6.4) to generate an object representing Wilkinson's polynomial from its roots to the available numerical precision; then find the roots of this representation of the polynomial.

9.3 Programming techniques and software development

9.3.1 General remarks

Commenting code

Throughout this book we have tried to comment the code examples and exercise solutions helpfully. This is a good practice, even for short scripts, but the effective use of comments is not an entirely trivial activity. Here is some general advice:

- Generally, prefer to place comments on their own lines rather than "inline" with code (that is, after but on the same line as the code they describe):

  ```
  # Volume of a dodecahedron of side length a
  V = (15 + 7 * np.sqrt(5)) / 4 * a**3
  ```

  rather than

  ```
  V = (15 + 7 * np.sqrt(5)) / 4 * a**3 # Volume of a dodecahedron of side a
  ```

- Explain *why* your code does what it does, don't simply explain *what* it does. Assume that the person reading your code knows the syntax of the language already. Thus,

  ```
  # Increase i by 10:
  i += 10
  ```

  is a terrible comment which adds nothing to the line of code it purports to explain. On the other hand,

  ```
  # Skip the next 10 data points
  i += 10
  ```

  at least gives some indication of the reason for the statement.
- Keep comments up-to-date with the code they explain. It is all too easy to change code without synchronizing the corresponding comments. This can lead to a situation that is worse than having no comment at all:

  ```
  # Skip the next 10 data points
  i += 20
  ```

  Which is correct? Is the comment correct in explaining the programmer's intention but the line of code buggy, or has the line of code been updated for some reason without changing the comment? If your code is likely to be subject to such changes, consider defining a separate variable to hold the change in `i`:

  ```
  DATA_SKIP = 10
  ...
  # Skip the next DATA_SKIP data points
  i += DATA_SKIP
  ```

  In fact, some programmers advocate aiming to minimize the number of comments by carefully choosing meaningful identifier names. For example, if we rename our index, we might even do away with the comment altogether:

  ```
  data_index += DATA_SKIP
  ```

- Explain functions carefully using docstrings. In Python, all functions have an attribute `__doc__` which is set to the docstring provided in the function definition (see Section 2.7.1). A docstring is usually a multiline, triple-quoted string providing an explanation of what the function does, the arguments it takes and the nature of its return value(s), if any. From an interactive shell, typing `help(`*`function_name`*`)` provides more detailed information concerning the function, including this docstring.

Example E9.5 An example of a well-commented function (to calculate the volume of a tetrahedron) is given here.

Listing 9.3 A function to calculate the volume of a tetrahedron

```
# eg9-tetrahedron.py
import numpy as np

def tetrahedron_volume(vertices=None, sides=None):
    """
    Return the volume of the tetrahedron with given vertices or side lengths.
    If vertices are given they must be in an array with shape (4,3): the
    position vectors of the four vertices in three dimensions; if the six sides are
    given, they must be an array of length 6. If both are given, the sides
    will be used in the calculation.

    Raises a ValueError if the vertices do not form a tetrahedron (e.g.,
    because they are coplanar, colinear or coincident).

    """

    # This method implements Tartaglia's formula using the Cayley-Menger
    # determinant:
    #             |0    1    1    1    1  |
    #             |1    0   s1^2 s2^2 s3^2|
    # 288 V^2 =   |1   s1^2  0   s4^2 s5^2|
    #             |1   s2^2 s4^2  0   s6^2|
    #             |1   s3^2 s5^2 s6^2  0  |
    # where s1, s2, ..., s6 are the tetrahedron side lengths.

    # Warning: this algorithm has not been tested for numerical stability.

    # The indexes of rows in the vertices array corresponding to all
    # possible pairs of vertices
    vertex_pair_indexes = np.array(((0, 1), (0, 2), (0, 3),
                                    (1, 2), (1, 3), (2, 3)))
    if sides is None:
        # If no sides were provided, work them out from the vertices
❶       vertices = np.asarray(vertices)
        if vertices.shape != (4,3):
            raise TypeError('vertices must be a numpy array with shape (4,3)')
        # Get all the squares of all side lengths from the differences between
        # the 6 different pairs of vertex positions
        vertex1, vertex2 = vertex_pair_indexes.T
        sides_squared = np.sum((vertices[vertex1] - vertices[vertex2])**2,
                               axis=-1)
```

```
    else:
        # Check that sides has been provided as a valid array and square it
        sides = np.asarray(sides)
        if sides.shape != (6,):
            raise TypeError('sides must be an array with shape (6,)')
        sides_squared = sides**2

    # Set up the Cayley-Menger determinant
    M = np.zeros((5,5))
    # Fill in the upper triangle of the matrix
    M[0,1:] = 1
    # The squared-side length elements can be indexed using the vertex
    # pair indexes (compare with the determinant illustrated above)
    M[tuple(zip(*(vertex_pair_indexes + 1)))] = sides_squared

    # The matrix is symmetric, so we can fill in the lower triangle by
    # adding the transpose
    M = M + M.T

    # Calculate the determinant and check it is positive (negative or zero
    # values indicate the vertices to not form a tetrahedron).
    det = np.linalg.det(M)
    if det <= 0:
        raise ValueError('Provided vertices do not form a tetrahedron')
    return np.sqrt(det / 288)
```

❶ Using `np.asarray` to convert `vertices` into a NumPy array if it isn't one already enables the function to work with any compatible object (such as a `list` of `lists`).

Style Guide for Python Code

The officially recommended coding conventions for Python are provided by a document known as PEP8 (available at www.python.org/dev/peps/pep-0008/). While it is acknowledged that it isn't always appropriate to follow these conventions all the time, Python programmers generally agree that they maximize the comprehensibility and maintainability of code. The focus is on consistency, readability and in minimizing the probability of hard-to-find typographical errors. Some of the highlights are

- Use *four spaces* per indentation level (and never tabs).[9]
- In assignments, put spaces around the = sign; for example, `a = 10`, not `a=10`.
- Use a maximum of 79 characters per line, where you need to split a line of code over more than one line:
  - favor implicit line continuation inside parentheses over the explicit use of the character, \ (see Section 2.3.1);
  - in arithmetic expressions, break around binary operators so that the new line is *after* the operator;
  - as far as possible, line up code so that expressions within parentheses line up.

[9] A good text editor can be configured to automatically expand tabs to a fixed number of spaces.

For example, the following is considered poor style:

```
lengthy_calculation = margin*margin_px + (border*border_px\
                             + padding*padding_px)
```

and might be better written as

```
lengthy_calculation = (margin*margin_px + (border*border_px +
                                         padding*padding_px))
```

- Separate top-level function and class definitions by two blank lines; within a class, separate them by one blank line.
- Use UTF-8 encoding for your source code (in Python 3 this is the default encoding anyway).
- Avoid wildcard imports (`from foo import *`).
- Separate operators from their operands with single spaces unless operations with different priorities are being combined; for example, write `x = x + 5` but `r2 = x**2 + y**2`.
- Don't use spaces around the `=` in keyword arguments; for example, in function calls use `foo(b=4.5)` not `foo(b = 4.5)`.
- Avoid putting more than one statement on the same line separated by semicolons; for example, instead of `a = 1; b = 2`, write `a, b = 1, 2` (see Section 4.3.1).
- *Functions*, *modules* and *packages* should have short, all-lowercase names. Use underscores in function and module names if necessary, but avoid them in package names.
- *Class names* should be in (upper) CamelCase, also known as CapWords; for example, `AminoAcid`, not `amino_acid`.
- Define *constants*[10] in all-capitals with underscores separating words; for example, `MAX_LINE_LENGTH`.

9.3.2 Editors

While, to some extent, the choice of text editor for writing code is a personal one, most programmers favor one with syntax highlighting and the possibility to define macros to speed up repetitive tasks. Popular choices include:

- Sublime Text, a commercial editor with per-user licensing and a free-evaluation option;
- Vim, a widely used cross-platform keyboard-based editor with a steep learning curve but powerful features. The more basic vi editor is installed on almost all Linux and Unix operating systems;
- Emacs, a popular alternative to Vim;
- Notepad++, a free Windows-only editor;
- SciTE, a fast, lightweight source code editor;
- Atom, another free, open-source, cross-platform editor.

[10] Note that Python doesn't really have constants in the same way that, for example, C does.

Beyond simple editors, there are fully featured integrated development environments (IDEs) that also provide debugging, code-execution, intelligent code-completion and access to operating system services. Here are some of the options available:

- Eclipse with the PyDev plugin, a popular free IDE;
- PyCharm, a cross-platform IDE with commercial and free editions;
- PythonAnywhere, an online Python environment with free and paid-for options (https://www.pythonanywhere.com/);
- Spyder, an open source IDE for scientific programming in Python, which integrates NumPy, SciPy, Matplotlib and IPython.

9.3.3 Version control

Unless properly managed, larger software projects (in practice, anything consisting of more than a single file of code) often rapidly descend into a tangle with modified versions, experimental code, ad hoc features and temporary files. The management of changes to the files comprising a software project is called *version control* (or *revision control*).

At its simplest, version control can involve simply keeping code in a number of parallel directories (folders), numbered chronologically as the software evolves. This approach can work, but if a small change in a large amount of code leads to a new version it is inefficient (a lot of unchanged code is copied across to the new directory). If a new version is created only when the code changes a lot, then there is scope for a lot of tangled code to be generated between versions.

To solve these problems, there are several version control software packages available, some of which are listed here. Most of these run as standalone applications on an operating system and can be invoked from the command line or used through a graphical interface. Some advantages are as follows:

- Many developers can collaborate on one project;
- *Branching*: the parallel development of two versions of the software at the same time, for example, to test out new features;
- *Tagging* (or *labeling*): a way of referring to a snapshot of the project in a particular state;
- Roll-back of a file in the project to a previous version;
- *Cloning*: a means of distributing a software project along with its history of changes;
- Some version control systems integrate with online repositories for storing and sharing code. The most famous of these is GitHub (https://github.com/).

We will not describe the working of version control systems in detail (the syntax varies between systems and there are extensive tutorials, documentation and even entire books written about each one). Some recommended options are:

- Git: the most widely adopted version control system, Git works on a *distributed* (or *decentralized*) basis, allowing developers to work on a project without sharing

a common network or central reference code repository. Open source projects can be hosted for free at *GitHub*.
http://git-scm.com/

- Mercurial: another distributed version control system.
http://mercurial.selenic.com/
- Subversion (SVN): a centralized option with free (for open source projects) hosting at SourceForge (http://sourceforge.net/). As Git has gained in popularity, SVN is not as widely used as it once was.
http://subversion.apache.org/

9.3.4 Unit tests

Unit testing is a way of validating software by focusing on individual units of source code. As an object-oriented programming language, for Python this usually means that individual classes (and sometimes even individual functions) are tested against a set of trial data (some of which may be deliberately incorrect or malformed). The aim is to catch any bugs which lead to the faulty interpretation of data. The set of unit tests also serve as a documented and verifiable assertion that the code does what it is supposed to. In some paradigms of code development, unit tests are written before the code itself.[11]

An important advantage of unit testing is that it provides a means of assuring that subsequent changes to the code (perhaps the addition of some functionality) does not break it: the upgraded code should pass the same unit tests that the original code did.

Unit testing your own code for a small project takes discipline. The tests are, themselves, computer code (and, perhaps, associated data) and need careful thought to write. The devising of suitable unit tests often prompts the programmer to think more deeply about the implementation of their code and can catch possible bugs before it is written.

Python's unit testing framework is based around the `unittest` module: a simple application is given in the example.

Example E9.6 Suppose we want to write a function to convert a temperature between the units Fahrenheit, Celsius and Kelvin (identified by the characters `'F'`, `'C'` and `'K'` respectively). The six formulas involved are not difficult to code, but we might wish to handle gracefully a couple of conditions that could arise in the use of this function: a physically unrealizable temperature (< 0 K) or a unit other than `'F'`, `'C'` or `'K'`.

Our function will first convert to Kelvin and then to the units requested; if the from-units and the to-units are the same for some reason, we want to return the original value unchanged. The function `convert_temperature` is defined in the file `temperature_utils.py`.

Listing 9.4 A function for converting between different temperature units

```
# temperature_utils.py

def convert_temperature(value, from_unit, to_unit):
    """ Convert and return the temperature value from from_unit to to_unit. """
```

[11] In particular, so-called 'extreme' programming.

```
    # Dictionary of conversion functions from different units *to* K
    toK = {'K': lambda val: val,
           'C': lambda val: val + 273.15,
           'F': lambda val: (val + 459.67)*5/9,
          }
    # Dictionary of conversion functions *from* K to different units
    fromK = {'K': lambda val: val,
             'C': lambda val: val - 273.15,
             'F': lambda val: val*9/5 - 459.67,
            }

    # First convert the temperature from from_unit to K
    try:
        T = toK[from_unit](value)
    except KeyError:
        raise ValueError('Unrecognized temperature unit: {}'.format(from_unit))

    if T < 0:
        raise ValueError('Invalid temperature: {} {} is less than 0 K'
                                  .format(value, from_unit))

    if from_unit == to_unit:
       # No conversion needed!
        return value

    # Now convert it from K to to_unit and return its value
    try:
        return fromK[to_unit](T)
    except KeyError:
        raise ValueError('Unrecognized temperature unit: {}'.format(to_unit))
```

To use the `unittest` module to conduct unit tests on the `convert_temperature`, we write a new Python script defining a class, `TestTemperatureConversion`, derived from the base `unittest.TestCase` class. This class defines methods that act as tests of the `convert_temperature` function. These test methods should call one of the base class's *assertion functions* to validate that the return value of `convert_temperature` is as expected. For example,

```
self.assertEqual(<returned value>, <expected value>)
```

returns `True` if the two values are exactly equal and `False` otherwise. Other assertion functions exist to check that a specific exception is raised (e.g., by invalid arguments) or that a returned value is `True`, `False`, `None`, and so on. The unit test code for our `convert_temperature` function is here.

Listing 9.5 Unit tests for the temperature conversion function

```
from temperature_utils import convert_temperature
import unittest

class TestTemperatureConversion(unittest.TestCase):

    def test_invalid(self):
        """
        There's no such temperature as -280 C, so convert_temperature should
        raise a ValueError.
```

```
        """
❶       self.assertRaises(ValueError, convert_temperature, -280, 'C', 'F')

    def test_valid(self):
        """ A series of valid temperature conversions to test. """

        test_cases = [((273.16, 'K',), (0.01, 'C')),
                      ((-40, 'C'), (-40, 'F')),
                      ((450, 'F'), (505.3722222222222, 'K'))]

        for test_case in test_cases:
            ((from_val, from_unit), (to_val, to_unit)) = test_case
            result = convert_temperature(from_val, from_unit, to_unit)
❷           self.assertAlmostEqual(to_val, result)

    def test_no_conversion(self):
        """
        Ensure that if the from-units and to-units are the same the
        temperature is returned exactly as it was passed and not converted
        to and from Kelvin, which may cause loss of precision.

        """
        T = 56.67
        result = convert_temperature(T, 'C', 'C')
❸       self.assertEqual(result, T)

    def test_bad_units(self):
        """ Check that ValueError is raised if invalid units are passed. """
        self.assertRaises(ValueError, convert_temperature, 0, 'C', 'R')
        self.assertRaises(ValueError, convert_temperature, 0, 'N', 'K')

unittest.main()
```

❶ `assertRaises` verifies that a specified exception is raised by the method `convert_temperature`. The necessary arguments to this method are passed after the method object itself.

❷ We need `assertAlmostEqual` here because the floating point arithmetic is likely to cause a loss of precision due to rounding errors.

❸ We use `assertEqual` here to ensure that the temperature value is returned as it was passed and not converted to and from Kelvin.

Running this script shows that our function passes its unit tests:

```
$ python eg9-temperature-conversion-unittest.py
...
----------------------------------------------------------------------
Ran 4 tests in 0.000s

OK
```

9.3.5 Further Reading

- F. Brooks, (1975, 1995). *The Mythical Man-Month*, Addison-Wesley. Near-legendary monograph on software development explaining why "adding man-power to a late software project makes it later."
- J. Loeliger and M. McCullough, (2012). *Version Control with Git*, O'Reilly.
- S. McConnell, (2004). *Code Complete: A Practical Handbook of Software Construction*, Microsoft Press.
- A. Hunt and D. Thomas, (1999). *The Pragmatic Programmer*, Addison-Wesley.

Appendix A Solutions

Answers to selected questions are given here. For further exercises and solutions, see scipython.com.

Q2.2.5 This question illustrates the danger of "wildcard" imports: the value of the variable `e=2` is replaced by the definition of `e` in the `math` module. The expression `d ** e` therefore raises 8 to the power of $e = 2.71828\cdots$ instead of squaring it.

Q2.2.7 Using Python's operators:

```
>>> a = 2
>>> b = 6
>>> 3 * (a**3*b - a*b**3) % 7
3
>>> a = 3
>>> b = 5
>>> 3 * (a**3*b - a*b**3) % 7
1
```

Q2.2.8 The thickness of the paper on the nth fold is $2^n t$, so we require $2^n t \geq d \Rightarrow n_{\mathrm{min}} = \lceil \log_2(d/t) \rceil$:

```
>>> d = 384400 * 1.e3        # distance to moon, m
>>> t = 1.e-4                # paper thickness, m
>>> math.log(d / t, 2)       # base-2 logarithm
41.805745474760016
```

Hence the paper must be folded 42 times to reach to the moon ($\lceil x \rceil$ denotes the *ceiling* of x: the smallest integer not less than x).

Q2.2.10 The `^` operator does not raise a number to another power (that is the `**` operator). It is the *bitwise xor* operator, and in binary `10^2` is `1010 xor 0010 = 1000`, which is `8` in decimal.

Q2.3.1 Slice the string `s='seehemewe'` as follows (other solutions are possible in some cases):

a. `s[:3]`
b. `s[3:5]`
c. `s[5:7]`
d. `s[7:]`
e. `s[3:6]`

f. `s[5:2:-1]`
g. `s[-2::-3]`

Q2.3.2 Simply slice the string backward and compare with the original:

```
>>> s = 'banana'
>>> s == s[::-1]
False
>>> s = 'deified'
>>> s == s[::-1]
True
```

Q2.3.5 This is not the correct way to test if the string `s` is equal to either `'ham'` or `'eggs'`. The expression `('eggs' or 'ham')` is a boolean one in which both arguments, being nonempty strings, evaluate to `True`. The expression short-circuits at the first `True` equivalent and this operand is returned (see Section 2.2.4): that is, `('eggs' or 'ham')` returns `'eggs'`. Because `s` is, indeed, the string `'eggs'` the equality comparison returns `True`. However, if the order of the operands is swapped, the boolean `or` again short-circuits at the first `True`-equivalent, which is now `'ham'` and returns it. The equality comparison with `s` fails, and the result is `False`.

There are two correct ways to test if `s` is one of two or more strings:

```
>>> s = 'eggs'
>>> s == 'ham' or s == 'eggs'
True
>>> s in ('ham', 'eggs')
True
```

(See Section 2.4.2 for more information about the syntax of the second statement.)

Q2.4.2 The problem is that `enumerate`, by default, returns the indexes and items of the array passed to it with the indexes starting at 0. The array passed to it is the slice `P[1:] = [5, 0, 2]` and so `enumerate` generates, in turn, the tuples `(0, 5)`, `(1, 0)` and `(2, 2)`. However, for our derivative we need the indexes into the original list, `P`, giving `(1, 5)`, `(2, 0)` and `(3, 2)`. There are two alternatives: pass the optional argument `start=1` to `enumerate` or add 1 to the default index:

```
>>> P = [4, 5, 0, 2]
>>> dPdx = []
>>> for i, c in enumerate(P[1:], start=1):
...     dPdx.append(i*c)
>>> dPdx
[5, 0, 6]
```

```
>>> P = [4, 5, 0, 2]
>>> dPdx = []
>>> for i, c in enumerate(P[1:]):
...     dPdx.append((i+1)*c)
>>> dPdx
[5, 0, 6]
```

Q2.4.3 Here is one solution:

```
>>> scores = [87, 75, 75, 50, 32, 32]
>>> ranks = []
>>> for score in scores:
...     ranks.append(scores.index(score) + 1)
...
>>> ranks
[1, 2, 2, 4, 5, 5]
```

Q2.4.4 The following calculates π to 10 decimal places.

```
>>> import math

>>> pi = 0
>>> for k in range(20):
...     pi += pow(-3, -k) / (2*k+1)   ❶
...
>>> pi *= math.sqrt(12)
>>> print('pi = ', pi)
pi =  3.1415926535714034
>>> print('error = ', abs(pi - math.pi))
error =  1.8389734179891093e-11
```

❶ The built-in `pow(x, j)` is equivalent to `(x)**j`.

Q2.4.5 `any(x) and not all(x)` is `True` if at least one item in `x` is equivalent to `True` but not all of them:

```
>>> x1, x2, x3 = [False, False], [1, 2, 3, 4], [1, 2, 3, 0]
>>> any(x1) and not all(x1)
False
>>> any(x2) and not all(x2)
False
>>> any(x3) and not all(x3)
True
```

Q2.4.6 Recall that the `*` operator unpacks a tuple into a positional argument list to a function. So if `z = zip(a,b)` is the (iterator) sequence: `(a0,b0), (a1, b1), (a2, b2), ...`. Unpacking this sequence in the call `zip(*z)` is equivalent to calling `zip` with these tuples as arguments:

`zip((a0, b0), (a1, b1), (a2, b2), ...)`

`zip` takes the first and second items from each tuple in turn, reproducing the original sequences:

`(a0, a1, a2, ...), (b0, b1, b2, ...)`

Q2.4.7 Simply zip the lists of sunshine hours and month names together and reverse-sort the resulting list of tuples:

```
>>> months = ['Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun',
...           'Jul', 'Aug', 'Sep', 'Oct', 'Nov', 'Dec']
>>> sun = [44.7, 65.4, 101.7, 148.3, 170.9, 171.4,
...        176.7, 186.1, 133.9, 105.4, 59.6, 45.8]
>>> for s, m in sorted(zip(sun, months), reverse=True):
...     print('{}: {:.1f} hrs'.format(m, s))
...
```

```
Aug: 186.1 hrs
Jul: 176.7 hrs
Jun: 171.4 hrs
May: 170.9 hrs
Apr: 148.3 hrs
Sep: 133.9 hrs
Oct: 105.4 hrs
Mar: 101.7 hrs
Feb: 65.4 hrs
Nov: 59.6 hrs
Dec: 45.8 hrs
Jan: 44.7 hrs
```

Q2.5.1 To normalize a list:

```
>>> a = [2,4,10,6,8,4]
>>> amin, amax = min(a), max(a)
>>> for i, val in enumerate(a):
...     a[i] = (val-amin) / (amax-amin)
...
>>> a
[0.0, 0.25, 1.0, 0.5, 0.75, 0.25]
```

Q2.5.2 The following code calculates Gauss's constant to 14 decimal places.

```
>>> import math
>>> tol = 1.e-14
>>> an, bn = 1., math.sqrt(2)
>>> while abs(an - bn) > tol:
...     an, bn = (an + bn) / 2, math.sqrt(an * bn)
...
>>> print('G = {:.14f}'.format(1/an))
G = 0.83462684167407
```

Q2.5.3 The following code produces the first 100 "fizzbuzz" numbers.

```
nmax = 100
for n in range(1, nmax+1):
    message = ''
    if not n % 3:
        message = 'fizz'
    if not n % 5:
        message += 'buzz'
❶   print(message or n)
```

❶ Note that if `n` is not divisible by either 3 or 5, `message` will be the empty string, which evaluates to `False` in this logical expression, so `n` is printed instead.

Q2.5.4 Here's one solution, using `stoich='C8H18'` as an example:

Listing A.1 The structural formula of a straight-chain alkane

```
# qn2-5-c-alkane-a.py

stoich = 'C8H18'

fragments = stoich.split('H')
nC = int(fragments[0][1:])
```

```
nH = int(fragments[1])
if nH != 2*nC + 2:
    print('{} is not an alkane!'.format(stoich))
else:
    print('H3C', end='')
    for i in range(nC-2):
        print('-CH2', end='')
    print('-CH3')
```

The output is:

```
H3C-CH2-CH2-CH2-CH2-CH2-CH2-CH3
```

Q2.7.1 Only (b) and (f) behave as intended:

a. In the absence of an explicit `return` statement, the `line` function returns `None`. Because `None` cannot be joined into a string, an error occurs:

```
my_sum = '\n'.join(['   56', '  +44', line, '  100', line])
...
TypeError: sequence item 2: expected str instance, NoneType found
```

b. This code works as intended.

c. The function `line` returns a string, as required, but is not called as `line()`: without the parentheses, `line` refers to the function object itself, which cannot be joined in a string, so an error occurs:

```
my_sum = '\n'.join(['   56', '  +44', line, '  100', line])
...
TypeError: sequence item 2: expected str instance, function found
```

d. This code does not cause an error, but outputs a string representation of the function instead of the string returned when the function is called:

```
   56
  +44
<function line at 0x103d9e9e0>
  100
<function line at 0x103d9e9e0>
```

e. This code generates unwanted `None` output:

```
   56
  +44
-----
None
  100
-----
None
```

This happens because the statement `print(line())` calls the function `line`, which prints a line of hyphens but also prints its return value (which is `None` since it doesn't return anything else explicitly).

f. This code works as intended.

Q2.7.2 The problem is within the `add_interest` function:

```
def add_interest(balance, rate):
    balance += balance * rate / 100
```

This creates a new `float` object, `balance`, *local* to the function, which is independent of the original `balance` object. When the function exits, the local `balance` is destroyed and the original `balance` never updated. One fix would be to `return` the updated balance value from the function:

```
>>> balance = 100
>>> def add_interest(balance, rate):
...     balance += balance * rate / 100
...     return balance
...
>>> for year in range(4):
...     balance = add_interest(balance, 5)
...     print('Balance after year {}: ${:.2f}'.format(year+1, balance))
...
Balance after year 1: $105.00
Balance after year 2: $110.25
Balance after year 3: $115.76
Balance after year 4: $121.55
```

Q2.7.3 The problem is that the function `digit_sum` does not return the sum of the digits of `n` that it has calculated. In the absence of an explicit `return` statement, a Python function returns `None`, but `None` isn't an acceptable object to use in a modulus calculation and so a `TypeError` is raised.

The fix is simply to add `return dsum`:

```
def digit_sum(n):
    """ Find and return the sum of the digits of integer n. """

    s_digits = list(str(n))
    dsum = 0
    for s_digit in s_digits:
        dsum += int(s_digit)
    return dsum

def is_harshad(n):
    return not n % digit_sum(n)
```

Now, as expected:

```
>>> is_harshad(21)
True
```

Q4.1.1 It is a good idea to keep the `try` block as small as possible to prevent exceptions that you do not want to catch being caught instead of the one you do. For example, in Example E4.5, suppose we read the file after opening it within the same `try` block:

```
try:
    fi = open(filename, 'r')
    lines = fi.readlines()
```

```
except IOError:
    ...
```

Now there are two errors that could give rise to an `IOError` Exception being raised: failure to open the file and failure to read its lines. The `except` clause is intended to handle the first case, but it will also be executed in the second case when it would be more appropriate to handle it differently (or leave it unhandled and stop program execution).

Q4.1.2 The point of `finally` in Example E4.5 is that statements in this block get executed *before* the function returns. If the line

```
print('   Done with file {}'.format(filename))
```

were moved to after the `try` block, it would not be executed if an `IOError` Exception is raised (because the function would have returned to its caller before this `print` statement is encountered.

Q4.2.1 This can easily be achieved with a `set`. Given the string, `s`:

```
set(s.lower()) >= set('abcdefghijklmnopqrstuvwxyz')
```

is `True` if it is a pangram. For example,

```
>>> s = 'The quick brown fox jumps over the lazy dog'
>>> set(s.lower()) >= set('abcdefghijklmnopqrstuvwxyz')
True
>>> s = 'The quick brown fox jumped over the lazy dog'
>>> set(s.lower()) >= set('abcdefghijklmnopqrstuvwxyz')
False
```

Q4.2.2 This function can be used to remove duplicates from an ordered list.

```
>>> def remove_dupes(l):
...     return sorted(set(l))
...
>>> remove_dupes([1,1,2,3,4,4,4,5,7,8,8,9])
[1, 2, 3, 4, 5, 7, 8, 9]
```

Note that although sets don't have an order, they are iterable and can be passed to the `sorted()` built-in method (which returns a `list`).

Q4.2.3 From within the Python interpreter:

```
>>> set('hellohellohello')
{'h', 'o', 'l', 'e'}
>>> set(['hellohellohello'])
{'hellohellohello'}
>>> set(('hellohellohello'))
{'h', 'o', 'l', 'e'}
>>> set(('hellohellohello',))
{'hellohellohello'}
>>> set(('hello', 'hello', 'hello'))
{'hello'}
>>> set(('hello', ('hello', 'hello')))
{'hello', ('hello', 'hello')}
>>> set(('hello', ['hello', 'hello']))
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'list'
```

Note the difference between initializing a `set` with a `list` of objects and attempting to add a `list` as an object in a set.

Q4.2.4 Note that the statement

```
>>> a |= {2,3,4,5}
```

does not change the `frozenset` but rather creates a new one from the union of the old one and the `set {2,3,4,5}`. (In the same way, we have seen that for `int` object `i`, the assignment `i = i + 1` rebinds the label `i` to a new integer object with value `i+1` rather than changing the value of the immutable `int` object previously bound to `i`.)

Q4.3.1 The list comprehension

```
>>> flist = [lambda x, i=i: x**i for i in range(4)]
```

creates the same list of anonymous functions as that in Example E4.10.

Note that we need to pass each `i` into the `lambda` function explicitly or else Python's closure rules will lead to every `lambda` function being equivalent to `x**3` (3 being the final value of `i` in the loop).

Q4.3.2 The code snippet outputs the first `nmax+1` rows of Pascal's Triangle:

```
[1]
[1, 1]
[1, 2, 1]
[1, 3, 3, 1]
[1, 4, 6, 4, 1]
[1, 5, 10, 10, 5, 1]
```

In the list comprehension assignment,

```
x = [([0]+x)[i] + (x+[0])[i] for i in range(n+1)]
```

the elements of two lists are added. The two lists are formed from the list representing the previous row by, in the first case, adding a `0` to the beginning of the list, and in the second case, by adding a `0` to the end of the list. In this way, the sum is taken over by neighboring pairs of numbers, with the end numbers unchanged. For example, if `x` is `[1, 3, 3, 1]`, the next row is formed by summing the elements in the lists

```
[0, 1, 3, 3, 1]
[1, 3, 3, 1, 0]
```

which yields the required `[1, 4, 6, 4, 1]`.

Q4.3.3

a. Index the items of `a` using the elements of `b`:

```
>>> [a[x] for x in b]
['E', 'C', 'G', 'B', 'F', 'A', 'D']
```

b. Index the items of `a` using the sorted elements of `b`. In this case, the returned list is just (a copy of) `a`:

```
>>> [a[x] for x in sorted(b)]
['A', 'B', 'C', 'D', 'E', 'F', 'G']
```

c. Index the items of `a` using the elements of `b` indexed at the elements of `b`(!)

```
>>> [a[b[x]] for x in b]
['F', 'G', 'D', 'C', 'A', 'E', 'B']
```

d. Associate each element of `b` with the corresponding element of `a` in a sequence of tuples: `[(4, 'A'), (2, 'B'), (6, 'C'), ...]`, which is then sorted – this method is used to return the elements of `a` corresponding to the ordered elements of `b`.

```
>>> [x for (y,x) in sorted(zip(b,a))]
['F', 'D', 'B', 'G', 'A', 'E', 'C']
```

Q4.3.4 To return a sorted list of (*key*, *value*) pairs from a dictionary:

```
>>> d = {'five': 5, 'one': 1, 'four': 4, 'two': 2, 'three': 3}
>>> d
{'four': 4, 'one': 1, 'five': 5, 'two': 2, 'three': 3}
>>> sorted([(k, v) for k, v in d.items()])
[('five', 5), ('four', 4), ('one', 1), ('three', 3), ('two', 2)]
```

Note that sorting the list of (*key*, *value*) tuples requires that the keys all have data types that can be meaningfully ordered. This approach will not work, for example, if the keys are a mixture of integers and strings since (in Python 3) there is no defined order to sort these types into: a `TypeError: unorderable types: int() < str()` exception will be raised.

To sort by *value* we could sort a list of (*value*, *key*) tuples, but to keep the returned list as (*key*, *value*) pairs, use

```
>>> sorted([(k, v) for k, v in d.items()], key=lambda item: item[1])
[('one', 1), ('two', 2), ('three', 3), ('four', 4), ('five', 5)]
```

The `key` argument to `sorted` specifies how to interpret each item in the list for ordering: here we want to order by the second entry `(item[1])` in each `(k, v)` tuple to order by value.

Q4.3.5 The following code encrypts (and decrypts) a telephone number held as a string using the "jump the 5" method.

```
''.join(['5987604321'[int(i)] if i.isdigit() else '-' for i in '555-867-5309'])
```

Q6.1.1 An `np.ndarray` is a NumPy class for representing multidimensional arrays in Python; in this book, we often refer to instances of this class simply as array objects. `np.array` is a function that constructs such objects from its arguments (usually a sequence).

Q6.1.2 To create a two-dimensional array, `array()` must be passed a *sequence of sequences* as a single argument: this call passes three sequence arguments instead. The correct call is

```
>>> np.array( ((1,0,0), (0,1,0), (0,0,1)) , dtype=float)
```

Q6.1.3 `np.array([0,0,0])` creates a one-dimensional array with three elements; `a = np.array([[0,0,0]])` creates a 1×3 two-dimensional array (i.e., `a[0]` is the one-dimensional array created in the first example).

Q6.1.4 Changing an array's type by setting `dtype` directly does not alter the data at the byte level, only how that data are interpreted as a number, string, and so on. As it happens, the byte-representations of zero are the same for integers (`int64`) and floats (`float64`), so the result of setting `dtype` is as expected. However, the 8-bytes representing `1.0` translate to the integer `4607182418800017408`. To convert the data type properly, use `astype()`, which returns a new array (with its own data):

```
In [x]: a = np.ones((3,))
In [x]: a
Out[x]: array([ 1.,  1.,  1.])

In [x]: a.astype('int')
In [x]: a
Out[x]: array([1, 1, 1])
```

Q6.1.5 Indexing and slicing a NumPy array:

a. `a[1,0,3]`
b. `a[0,2,:]` (or just `a[0,2]`)
c. `a[2,...]` (or `a[2,:,:]` or `a[2]`)
d. `a[:,1,:2]`
e. `a[2,:,1:-1]` ("in the third block, for each row take the items in the middle two columns").
f. `a[:,::-1,0]` ("for each block, traverse the rows backward and take the item in the first column of each").
g. Defining the three 2×2 index arrays for the blocks, rows and columns locating our elements as follows:

```
ia = np.array([[0, 0], [2, 2]])
ja = np.array([[0, 0], [3, 3]])
ka = np.array([[0, 3], [0, 3]])
```

`a[ia,ja,ka]` returns the desired result.

Q6.1.6 For example,

```
In [a]: a = np.array([0, -1, 4.5, 0.5, -0.2, 1.1])
In [x]: a[abs(a)<=1]
Out[x]: array([ 0. , -1. ,  0.5, -0.2])
```

Q6.1.7 In the following code:

```
In [x]: a, b = -2.00231930436153, -2.0023193043615
In [x]: np.isclose(a, b, atol=1.e-14)
Out[x]: True
```

`np.isclose()` returns `True` because although the absolute difference between the two numbers is greater than 10^{-14}, it is (significantly) less than `rtol * abs(b)`, the contribution from the default *relative* difference. To obtain the expected behavior, set `rtol` to 0:

```
In [x]: np.isclose(-2.00231930436153, -2.0023193043615, atol=1.e-14, rtol=0)
Out[x]: False
```

Q6.1.8 The different behavior here is due to the finite precision with which real numbers are stored: double-precision floating point numbers are only represented to the equivalent of about 15 decimal places and so the two numbers being compared here are the same to within this precision:

```
In [x]: 3.1415926535897932 - 3.141592653589793
Out[x]: 0.0
```

Q6.1.9 For example,

```
In [x]: N = 5
In [x]: Nsq = N**2
In [x]: np.allclose(np.sort(magic_square.flatten()),
                    np.linspace(1, Nsq, Nsq).astype(int))
Out[x]: True

In [x]: Nsum = N * (N**2 + 1) // 2
In [x]: np.allclose(np.sum(magic_square, axis=0), Nsum)
Out[x]: True

In [x]: np.allclose(np.sum(magic_square, axis=1), Nsum)
Out[x]: True

In [x]: n.allclose(np.diag(magic_square), Nsum)
Out[x]: True

❶ In [x]: n.allclose(np.diag(np.fliplr(magic_square)), Nsum)
Out[x]: True
```

❶ `np.fliplr` flips the array in the left/right direction. An alternative way to get this “other” diagonal is with `a.ravel()[N-1:-N+1:N-1]`.

Q6.1.10 The following statement will determine if a sequence `a` is increasing or not:

```
np.all(np.diff(a) > 0)
```

Q6.1.11 In the first case, a single object is created of the requested `dtype` and multiplied by a scalar (regular Python `int`). Python “upcasts” to return the result in `dtype` that can hold it:

```
In [x]: x = np.uint8(250)
In [x]: type(x*2)
Out[x]: numpy.int64
```

However, a `ndarray`, because it has a fixed byte size, cannot be upcast in the same way: its own `dtype` takes precedence over that of the scalar multiplying it, and so the multiplication is carried out modulo 256.

Compare this with the result of multiplying two scalars with the same `dtype`:

```
In [x]: np.uint8(250) * np.uint8(2)
Out[x]: 244            # (of type np.uint8)
```

(You may also see a warning: `RuntimeWarning: overflow encountered in ubyte_scalars`.)

Q6.4.1 The `Polynomial deriv` method returns a `Polynomial` object (in this case with a single term, the coefficient of x^0, equal to `18`). This object is not equal to the integer object with value `18`.

Q6.4.2 Using `numpy.polynomial.Polynomial`,

```
In [x]: p1 = Polynomial([-11,1,1])
In [x]: p2 = Polynomial([-7,1,1])
In [x]: p = p1**2 + p2**2
In [x]: dp = p.deriv()      # first derivative
In [x]: stationary_points = dp.roots()
In [x]: ddp = dp.deriv()     # second derivative
In [x]: minima = stationary_points[ddp(stationary_points) > 0]
In [x]: maxima = stationary_points[ddp(stationary_points) < 0]
In [x]: inflections = stationary_points[np.isclose(ddp(stationary_points),0)]
In [x]: print(np.array((minima, p(minima))).T)
[[-3.54138127  8.        ]
 [ 2.54138127  8.        ]]
In [x]: print(np.array((maxima, p(maxima))).T)
[[  -0.5   ,  179.125]]
In [x]: print(np.array((inflections, p(inflections))).T)
[]
```

That is, the function has two minima,

$$f(-3.54138127) = 8$$
$$f(2.54138127) = 8$$

one maximum,

$$f(-0.5) = 179.125$$

and no points of inflection / undulation.

Q6.5.1 Without overcomplicating things,

```
In [x]: pauli_matrices = np.array((
                                    ((0, 1), (1, 0)),
                                    ((0, -1j), (1j, 0)),
                                    ((1, 0), (0, -1))
                                   ))
In [x]: I2 = np.eye(2)
In [x]: for sigma in pauli_matrices:
   ...:     print(np.allclose(sigma.T.conj().dot(sigma), I2))
True
True
True
```

Q6.5.2 The following code fits the coefficients to the required quadratic equation. Note that this is a *linear* least squares fit even though the function is nonlinear in time because it is linear with respect to the coefficients.

```
# qn6-9-b-quadratic-fit-a.py
import numpy as np
```

```
import pylab
Polynomial = np.polynomial.Polynomial

x = np.array([1.3, 6.0, 20.2, 43.9, 77.0, 119.6, 171.7, 233.2, 304.2,
              384.7, 474.7, 574.1, 683.0, 801.3, 929.2, 1066.4, 1213.2,
              1369.4, 1535.1, 1710.3, 1894.9])
dt, n = 0.1, len(x)
tmax = dt * (n-1)
t = np.linspace(0, tmax, n)

A = np.vstack((np.ones(n), t, t**2)).T
coefs, resid, _, _ = np.linalg.lstsq(A, x)

# Initial position (cm) and speed (cm.s-1), acceleration due to gravity (m.s-2)
x0, v0, g = coefs[0], coefs[1], coefs[2] * 2 / 100

print('x0 = {:.2f} cm, v0 = {:.2f} cm.s-1, g = {:.2f} m.s-2'.format(x0, v0, g))

xfit = Polynomial(coefs)(t)
pylab.plot(t, x, 'ko')
pylab.plot(t, xfit, 'r')
pylab.xlabel('Time (sec)')
pylab.ylabel('Distance (cm)')
pylab.show()
```

The fitted function is shown in Figure A.1.

Q6.7.1 The first case,

```
In [x]: a = np.array([6,6,6,7,7,7,7,7,7])
In [x]: a[np.random.randint(len(a), size=5)]
array([7, 7, 7, 6, 7])      # (for example)
```

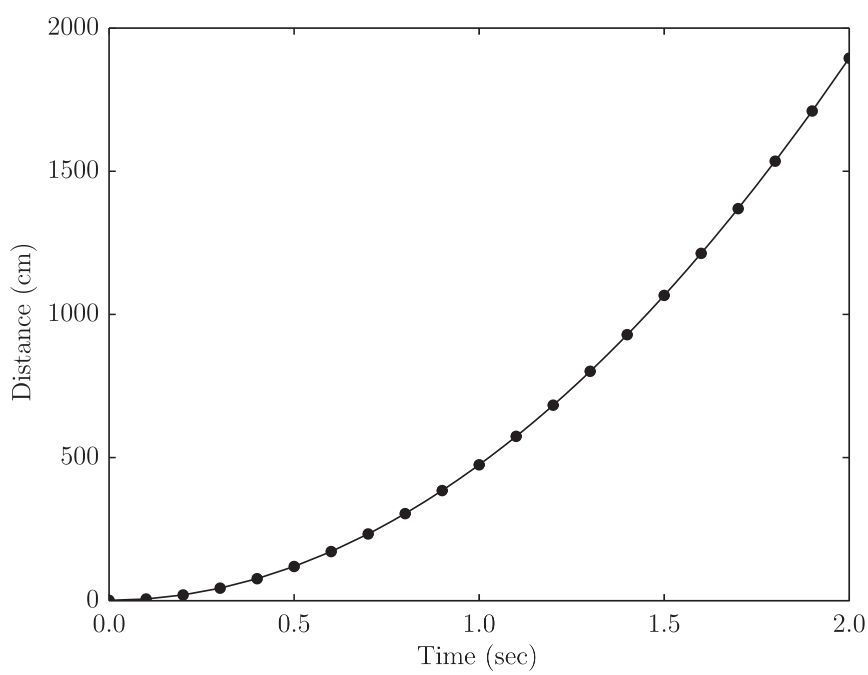


Figure A.1 Least squares fit to the function $x = x_0 + v_0 t + \frac{1}{2} g t^2$.

Samples randomly from the array `a` with replacement: for each item selected the probability of a 6 is $\frac{1}{3}$ and the probability of a 7 is $\frac{2}{3}$.

In the second case,

```
In [x]: np.random.randint(6, 8, 5)
array([6, 6, 7, 7, 7])       # (for example)
```

the numbers are drawn from [6, 7] uniformly, so the probabilities of each number being selected is $\frac{1}{2}$.

Q6.7.2 The function `np.random.randint` samples uniformly from the half-open interval, `[low, high)`, so to get the equivalent behavior to `np.random.random_integers` in Example E6.16 we need:

```
In [x]: a, b, n = 0.5, 3.5, 4
In [x]: a + (b-a) * (np.random.randint(1, n+1, size=10) - 1) / (n-1)
Out[x]: array([ 0.5,  1.5,  0.5,  3.5,  1.5,  3.5,  2.5,  0.5,  1.5,  1.5])
```

Q6.7.3 The probability of winning is one in

$$\binom{75}{5}\binom{15}{1} = \frac{75 \cdot 74 \cdot 73 \cdot 72 \cdot 71}{1 \cdot 2 \cdot 3 \cdot 4 \cdot 5} \cdot 15 = 258890850$$

To pick five random numbers from 1–75 and one from 1–15:

```
In [x]: (sorted(np.random.choice(np.arange(1,76), 5, replace=False)),
         np.random.randint(15)+1)
([4, 21, 35, 36, 64], 14)
```

Q6.7.4 Here is a more general solution to the problem. Draw the distribution of misprints across the book from the binomial distribution using `np.random.binomial` and count up how many pages have more than q misprints on them. To compare with the Poisson distribution, for the number of misprints on a page, X, we must calculate $\Pr(X >= q) = 1 - \Pr(X < q) = 1 - (\Pr(X = 0) + \Pr(X = 1) + \cdots + \Pr(X = q - 1)$:

Listing A.2 Calculating the probability of q or more misprints on a given page of a book.

```
# qn6-7-d-misprints-a.py
import numpy as np

n, m = 500, 400
q = 2
ntrials = 100
errors_per_page = np.random.binomial(m, 1/n, (ntrials, n))
av_ge_q = np.sum(errors_per_page>=q) / n / ntrials
print('Probability of {} or more misprints on a given page'.format(q))
print('Result from {} trials using binomial distribution: {:.6f}'
            .format(ntrials, av_ge_q))

# Now calculate the same quantity using the Poisson approximation,
# Pr(X>=q) = 1 - exp(-lam)[1 + lam + lam^2/2! + ... + lam^(q-1}/(q-1)!]
lam = m/n
poisson = 1
term = 1
for k in range(1,q):
```

```
    term *= lam/k
    poisson += term
poisson = 1 - np.exp(-lam) * poisson
print('Result from Poisson distribution: {:.6f}'.format(poisson))
```

A sample output is

```
Probability of 2 or more misprints on a given page
Result from 100 trials using binomial distribution: 0.190200
Result from Poisson distribution: 0.191208
```

Q6.8.1 The two methods for calculating the DFT can be timed using the IPython `%timeit` magic function

```
In [x]: import numpy as np
In [x]: n = 512
In [x]: # Our input function is just random numbers
In [x]: f = np.random.rand(n)

In [x]: # Time the NumPy (Cooley-Tukey) DFT algorithm
In [x]: %timeit np.fft.fft(f)
100000 loops, best of 3: 13.1 us per loop

In [x]: # Now calculate the DFT by direct summation
In [x]: k = np.arange(n)
In [x]: m = k.reshape((n, 1))
In [x]: w = np.exp(-2j * np.pi * m * k / n)
In [x]: %timeit np.dot(w, f)
1000 loops, best of 3: 354 us per loop

In [x]: # Check the two methods produce the same result
In [x]: ftfast = np.fft.fft(f)
In [x]: ftslow = np.dot(w, f)
In [x]: np.allclose(ftfast, ftslow)
Out[x]: True
```

The Cooley-Tukey algorithm is found to be almost 30 times faster than the direct method. In fact, this algorithm can be shown to scale as $\mathcal{O}(n \log n)$ compared with $\mathcal{O}(n^2)$ for direct summation.

Q8.1.1 Simply change the line:

```
for rec in constants[-10:]:
```

to:

```
for rec in constants[constants['rel_unc'] > 0][:10]:
```

The most accurately known constant is the electron g-factor.

```
2.64693e-07 ppm: electron g factor = -2.00232
2.69687e-07 ppm: electron mag. mom. to Bohr magneton ratio = -1.00116
3.7956e-06 ppm: electron magn. moment to Bohr magneton ratio = -1.00116
4.96096e-06 ppm: atomic unit of time = 2.41888e-17 s
...
```

Q8.1.2 The calculation $N/V = p/k_{\mathrm{B}}T$ for the stated conditions can be done entirely with constants from `scipy.constants`:

```
In [x]: scipy.constants.atm / scipy.constants.k / scipy.constants.zero_Celsius
Out[x]: 2.686780501003883e+25
```

This is the *Loschmidt constant* which is defined by the 2010 CODATA standards and included in `scipy.constants` (see the documentation for details):

```
In [x]: from scipy import constants
In [x]: constants.value('Loschmidt constant (273.15 K, 101.325 kPa)')
Out[x]: 2.6867805e+25
```

Q8.2.1 By numerical integration, the result is seen to be 3:

```
In [x]: from scipy.integrate import quad
In [x]: import numpy as np
In [x]: func = lambda x: np.floor(x) - 2*np.floor(x/2)
In [x]: quad(func,0,6)
Out[x]: (2.999964948683555,0.0009520766614606472)
```

Q8.2.2 In the following we assume the following imports:

```
In [x]: import numpy as np
In [x]: from scipy.integrate import quad
```

a.

```
In [x]: f1 = lambda x: x**4 * (1 - x)**4/(1 + x**2)

In [x]: quad(f1, 0, 1)
Out[x]: (0.0012644892673496185, 1.1126990906558069e-14)

In [x]: 22/7 - np.pi
Out[x]: 0.0012644892673496777
```

b.

```
In [x]: f2 = lambda x: x**3/(np.exp(x) - 1)

In [x]: quad(f2, 0, np.inf)
Out[x]: (6.49393940226683, 2.628470028924825e-09)

In [x]: np.pi**4 / 15
Out[x]: 6.493939402266828
```

c.

```
In [x]: f3 = lambda x: x**-x

In [x]: quad(f3, 0, 1)
Out[x]: (1.2912859970626633, 3.668398917966442e-11)

In [x]: np.sum(n**-n for n in range(1,20))
Out[x]: 1.2912859970626636
```

d.

```
In [x]: from scipy.misc import factorial
In [x]: f4 = lambda x, p: np.log(1/x)**p
In [x]: for p in range(10):
   ...:     print(quad(f4, 0, 1, args=(p,))[0], factorial(p))
   ...:
1.0 1.0
0.9999999999999999 1.0
1.9999999999999991 2.0
6.000000000000064 6.0
```

```
24.000000000000014 24.0
119.9999999999327 120.0
719.9999999989705 720.0
5039.99999945767 5040.0
40320.00000363255 40320.0
362880.00027390465 362880.0
```

e.
```
In [x]: from scipy.special import i0
In [x]: z = np.linspace(0,2,100)
In [x]: y1 = i0(z)
In [x]: f5 = lambda theta, z: np.exp(z*np.cos(theta))
In [x]: y2 = np.array([quad(f5, 0, 2*np.pi, args=(zz,))[0] for zz in z])
In [x]: y2 /= 2 * np.pi
In [x]: np.max(abs(y2-y1))
Out[x]: 3.4796610037801656e-12
```

Q8.2.3 To estimate π by integration of the constant function $f(x, y) = 4$ over the quarter circle with unit radius in the quadrant $x > 0, y > 0$:

```
In [x]: from scipy.integrate import dblquad
In [x]: dblquad(lambda y, x: 4, 0, 1, lambda x: 0, lambda x: np.sqrt(1-x**2))
Out[x]: (3.1415926535897922, 3.533564552071766e-10)
```

Q8.2.4 The integral to be calculated is

$$\int_0^1 \int_0^{2\pi} r \,\mathrm{d}\theta \,\mathrm{d}r = \pi .$$

Note that the inner integral is over θ and the outer is over r. Therefore, the call to `dblquad` should call the function $f(r, \theta) = r$ as `lambda theta, r: r` (note the order of the arguments).

```
In [x]: dblquad(lambda theta, r: r, 0, 1, lambda r: 0, lambda r: 2*np.pi)
Out[x]: (3.141592653589793, 3.487868498008632e-14)
```

Alternatively, swap the order of the integration:

```
dblquad(lambda r, theta: r, 0, 2*np.pi, lambda theta: 0, lambda theta: 1)
(3.141592653589793, 3.487868498008632e-14)
```

Q8.4.1 Rewrite the equation as

$$f(x) = x + 1 + (x - 3)^{-3} = 0.$$

This function is readily plotted and the roots may be bracketed in $(-2, -0.5)$ and $(0, 2.99)$ (avoiding the singularity at $x = 3$).

```
In [x]: f = lambda x: x + 1 + (x-3)**-3
In [x]: brentq(f, -2, -0.5), brentq(f, 0, 2.99)
Out[x]: (-0.984188231211512, 2.3303684533047426)
```

Q8.4.2 Some examples of root-finding for which the Newton-Raphson algorithm fails and how to solve this.

a.
```
In [x]: newton(lambda x: x**3 - 5*x, 1, lambda x: 3*x**2 - 5)
...
RuntimeError: Failed to converge after 50 iterations, value is 1.0
```

The Newton-Raphson algorithm enters an endless cycle of values for x:

$$\begin{aligned}
x_0 = 1: &\quad x_1 = x_0 - f(x_0)/f'(x_0) = -1 \\
x_1 = -1: &\quad x_2 = x_1 - f(x_1)/f'(x_1) = 1 \\
x_2 = 1: &\quad x_3 = x_2 - f(x_2)/f'(x_2) = -1 \\
\ldots &
\end{aligned}$$

Alternative starting points converge correctly on a root. Even a very small displacement from $x = 0$ ensures convergence:

```
In [x]: newton(lambda x: x**3 - 5*x, 1.0001, lambda x: 3*x**2 - 5)
Out[x]: 2.23606797749979
In [x]: newton(lambda x: x**3 - 5*x, 1.1, lambda x: 3*x**2 - 5)
Out[x]: -2.23606797749979
In [x]: newton(lambda x: x**3 - 5*x, 0.5, lambda x: 3*x**2 - 5)
Out[x]: 0.0
```

b.
```
In [x]: f, fp = lambda x: x**3 - 3*x+1, lambda x: 3*x**2 - 3
In [x]: newton(f, 1, fp)
Out[x]: 1.0
In [x]: f(1.0)
Out[x]: -1
```

The algorithm converged, but not on a root! Unfortunately, the gradient of the function is zero at the chosen starting point and because of round-off error this has not led to a `ZeroDivisionError`. To find the roots, choose different starting points such that $f'(x_0) \neq 0$, or use a different method after bracketing the roots by inspection of a plot of the function:

```
In [x]: brentq(f, -0.5, 0.5), brentq(f, -2, -1.5), brentq(f, 1, 2)
Out[x]: (0.34729635533386066, -1.879385241571423, 1.532088886237956)
```

c. The function $f(x) = 2 - x^5$ has a flat plateau around $f(0) = 2$ and the small gradient here leads to slow convergence on the root:

```
In [x]: newton(f, 0.01, fp)
...
RuntimeError: Failed to converge after 50 iterations, value is ...
```

To find it using `newton` either move the starting point closer to the root, or increase the maximum number of iterations:

```
In [x]: newton(f, 0.01, fp, maxiter=100)
Out[x]: 1.148698354997035
```

d. This is another example of a function that generates an endless cycle of values from the Newton-Raphson method:

```
In [x]: f = lambda x: x**4 - 4.29 * x**2 - 5.29
In [x]: fp = lambda x: 4*x**3 - 8.58 * x
In [x]: newton(f, 0.8, fp)
...
RuntimeError: Failed to converge after 50 iterations, value is ...
```

Unlike the function in (a), the region $0.6 \leq x_0 \leq 1.1$ *attracts* this cyclic behavior, so one needs to initialize the algorithm outside this range to obtain the roots ± 2.3. For example,

```
In [x]: newton(f, 1.2, fp)
Out[x]: -2.3
```

Q8.4.3 In general, there are two (physically distinct) possible angles θ_0 corresponding to the projectile passing through the specified point, $(x_1, y_1) = (5, 15)$, on the way up or on the way down. These values are the roots in $(0, \pi/2)$ of the function

$$f(\theta_0; x_1, z_1) = x_1 \tan\theta_0 - \frac{gx_1^2}{2v_0^2 \cos^2\theta_0} - z_1$$

After bracketing the roots with a rough plot of $f(\theta_0)$, we can use `brentq`:

```
In [x]: g = 9.81
In [x]: v0, x1, z1 = 25, 5, 15
In [x]: f = lambda theta0, x1, z1: x1 * np.tan(theta0) - g / 2\
                                  * (x1 / v0 / np.cos(theta0))**2 - z1
In [x]: th1 = brentq(f, 1, 1.4, args=(x1,z1))
In [x]: th2 = brentq(f, 1.5, 1.6, args=(x1,z1))
In [x]: np.degrees(th1), np.degrees(th2)
Out[x]: (74.172740936822834, 87.392310240255171)
```

That is, $\theta_0 = 74.2°$ or $\theta_0 = 87.4°$.

Q9.1.1 Let $x = 0.9999\cdots$. Then,

$$10x = 9.9999\cdots = 9 + x \;\Rightarrow\; 9x = 9 \;\Rightarrow\; x = 1.$$

Q9.1.2 This occurs because `math.pi` is only a (double-precision floating point) approximation to π, and the tangent of this approximate value happens to be negative:

```
In [x]: math.tan(math.pi)
Out[x]: -1.2246467991473532e-16
```

Taking the square root leads to the math domain error.

Q9.1.3 The problem, of course, is that the expression has been written using double-precision floating point numbers and the difference between the sum of the first two terms and the third is smaller than the precision of this representation. Using the exact representation in integer arithmetic,

```
In [x]: 844487**5 + 1288439**5
Out[x]: 3980245235185639013055619497406
In [x]: 1288439**5
Out[x]: 3980245235185639013290924656032
```

giving a difference of

```
In [x]: 844487**5 + 1288439**5 - 1318202**5
Out[x]: -235305158626
```

The finite precision of the floating point representation used, however, truncates the decimal places before this difference is apparent:

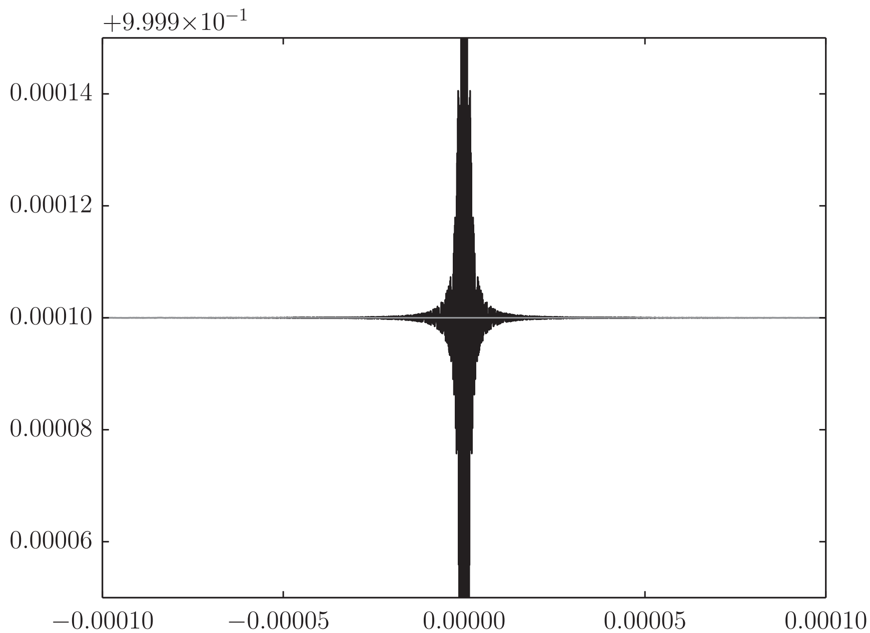


Figure A.2 A comparison of the numerical behavior of $f(x) = (1 - \cos^2 x)/x^2$ and $g(x) = \sin^2 x/x^2$ close to $x = 0$.

```
In [x]: 844487.**5 + 1288439.**5
Out[x]: 3.980245235185639e+30
In [x]: 1318202.**5
Out[x]: 3.980245235185639e+30
```

This is an example of *catastrophic cancellation*.

Q9.1.4 The expression `1 - np.cos(x)**2` suffers from catastrophic cancellation close to `x=0` resulting in a dramatic loss of precision and wild oscillations in the plot of $f(x)$ (Figure A.2). Consider, for example, `x = 1.e-9`: in this case, the *difference* `1 - np.cos(x)**2` is indistinguishable from zero (at double precision) so `f(x)` returns 0. Conversely, `np.sin(x)**2` is indistinguishable from `x**2` and `g(x)` returns `1.0` correctly.

Listing A.3 A comparison of the numerical behavior of $f(x) = (1 - \cos^2 x)/x^2$ and $g(x) = \sin^2 x/x^2$ close to $x = 0$.

```
# qn9-1-c-cos-sin-a.py

import numpy as np
import pylab

f = lambda x: (1 - np.cos(x)**2)/x**2
g = lambda x: (np.sin(x)/x)**2

x = np.linspace(-0.0001, 0.0001, 10000)

pylab.plot(x, f(x))
pylab.plot(x, g(x))
```

```
pylab.ylim(0.99995, 1.00005)
pylab.show()
```

Q9.1.5 We cannot compare with `==` because `nan` is not equal to itself. However, it is the *only* floating point number that is not equal to itself, so use `!=` instead:

```
In [x]: c = 0 * 1.e1000     # 0 * inf is nan
In [x]: c != c
Out[x]: True                # c isn't equal to itself, so must be nan
```